ecma INTERNATIONAL

Standard ECMA-372

1st Edition / December 2005

C++/CLI Language Specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

Standard

ECMA-372

1st Edition / December 2005

C++/CLI Language Specification

Table of Contents

Introduction

This Standard is based on a submission from Microsoft. It describes a technology, called C++/CLI, which is a binding between the Standard C++ programming language and the Common Language Infrastructure (CLI). That submission evolved from another Microsoft project, *Managed Extensions for C++*, the first widely distributed implementation of which was released by Microsoft in July 2000, as part of its .NET Framework initiative. The first widely distributed beta implementation of C++/CLI was released by Microsoft in July 2004.

Ecma Technical Committee 39 (TC39) Task Group 5 (TG5) was formed in October 2003, to produce a standard for C++/CLI. (Another Task Group, TG3, was formed in September 2000 to produce a standard for a library and execution environment called Common Language Infrastructure. The current version of that standard is ECMA-335, 3rd edition, June 2005. CLI is based on a subset of the .NET Framework.)

The goals used in the design of C++/CLI were as follows:

- Provide an elegant and uniform syntax and semantics that give a natural feel for C++ programmers.
- Provide first-class support for CLI features (e.g., properties, events, garbage collection, and generics) for all types including existing Standard C++ classes.
- Provide first-class support for Standard C++ features (e.g., deterministic destruction, templates) for all types including CLI classes.
- Preserve the meaning of existing Standard C++ programs by specifying pure extensions wherever possible.

The development of this standard started in December 2003.

It is expected there will be future revisions to this standard, primarily to add new functionality.

1. Scope

This Standard specifies requirements for implementations of the C++/CLI binding. The first such requirement is that they implement the binding, and so this Standard also defines C++/CLI. Other requirements and relaxations of the first requirement appear at various places within this Standard.

C++/CLI is an extension of the C++ programming language as described in ISO/IEC 14882:2003, *Programming languages — C++*. In addition to the facilities provided by C++, C++/CLI provides additional keywords, classes, exceptions, namespaces, and library facilities, as well as garbage collection.

2. Conformance

Clause §1.4, “Implementation compliance”, of the C++ Standard applies to this Standard.

3. Normative references

The following normative documents contain provisions, which, through reference in this text, constitute provisions of this Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ECMA-335, 3rd edition, June 2005, *Common Language Infrastructure (CLI)*, all Partitions and the accompanying library XML.

ISO/IEC 2382.1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*.

ISO/IEC 10646 (all parts), *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

ISO/IEC 14882:2003, *Programming languages — C++*. [*Note*: Revision of the C++ Standard is currently underway, and changes proposed in that revision will affect future versions of this C++/CLI standard. For an example, see §9.1.1. *end note*]

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989). (This standard is widely known by its U.S. national designation, ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*.)

This Standard supports the same version of Unicode as the CLI standard.

4. Definitions

For the purposes of this Standard, the following definitions apply. Other terms are defined where they appear in ***italic*** type or on the left side of a syntax rule. Terms explicitly defined in this Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this Standard are to be interpreted according to the C++ Standard, ISO/IEC 14882:2003.

application — Refers to an assembly that has an entry point. When an application is run, a new application domain is created. Several different instantiations of an application can exist on the same machine at the same time, and each has its own application domain.

application domain — An entity that enables application isolation by acting as a container for application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. A type loaded into one application domain is distinct from the same type loaded into another application domain, and objects on the CLI heap are not directly shared between application domains. Each application domain has its own copy of static variables for these types, and a static constructor for a type is run at most once per application domain. Implementations are free to provide implementation-specific policy or mechanisms for the creation and destruction of application domains.

assembly — Refers to one or more files that are output by the compiler as a result of program compilation. An assembly is a configured set of loadable code modules and other resources that together implement a unit of functionality. An assembly can contain types, the executable code used to implement these types, and references to other assemblies. The physical representation of an assembly is defined by the CLI Standard (§3). Essentially, an assembly is the output of the compiler. An assembly that has an entry point is called an application. (See also “metadata”.)

attribute — A characteristic of a type and/or its members that contains descriptive information. While the most common attributes are predefined, and have a specific encoding in the metadata associated with them, user-defined attributes can also be added to the metadata.

boxing — An explicit or implicit conversion from any value class type `V` to type `V^`, in which a `V` box is allocated on the CLI heap, and the value is copied into that box. (See also “unboxing”.)

CIL — Common Intermediate Language, the instruction set of the Virtual Execution System. This instruction set is defined in Partition III of the CLI Standard (§3).

CLI array — A CLI-specific array. A Standard C++-style array is referred to as a ***native array*** or, more simply, ***array***, whenever the distinction is needed. A CLI array differs from a native array in that the former is allocated on the CLI heap, and can have a rank other than one.

CLS compliance — The Common Language Specification (CLS) defines language interoperability rules, which apply only to items that are visible outside of their defining assembly. CLS compliance is described in Partition I of the CLI Standard (§3).

definition, out-of-class — A synonym for what Standard C++ calls a “non-inline definition”.

delegate — A ref class such that an instance of it can encapsulate one or more functions. Given a delegate instance and an appropriate set of arguments, one can invoke all of that delegate instance’s functions with that set of arguments.

event — A member that enables a class or a CLI object to provide notifications.

field — A synonym for what Standard C++ calls a “data member”.

function, abstract — A synonym for what Standard C++ calls a “pure virtual function”.

garbage collection — The process by which memory allocated from the CLI heap is automatically reclaimed on the CLI heap.

gc-lvalue — An expression that refers to an entity that might be allocated on the CLI heap. (See also "lvalue".)

handle — A handle is called an "object reference" in the CLI specification. For any CLI class type `T`, the declaration `T^ h` declares a handle `h` to type `T`, where the object to which `h` is capable of pointing resides on the CLI heap. A handle tracks, is rebindable, and can point to a whole object only. (See also "type, reference, tracking".)

heap, CLI — The storage area (accessed by `gcnew`) that is under the control of the garbage collector of the Virtual Execution System as specified in the CLI. (See also "heap, native".)

heap, native — The storage area (accessed by `new`) as defined in the C++ Standard (§18.4). (See also "heap, CLI".)

instance — An instance of a type.

lvalue — This has the same meaning as that defined in the C++ Standard (§3.10). (See also "gc-lvalue".)

metadata — Data that describes and references the types defined by the Common Type System (CTS). Metadata is stored in a way that is independent of any particular programming language. Thus, metadata provides a common interchange mechanism for use between tools that manipulate programs (such as compilers and debuggers) as well as between these tools and the Virtual Execution System.

pinning — The process of (temporarily) keeping constant the location of an object that resides on the CLI heap, so that object's address can be taken with that address remaining constant.

property — A member that defines a named value and the functions that access that value. A property definition defines the accessing contracts on that value. Hence, the property definition specifies the accessing functions that exist and their respective function contracts.

rebinding —The act of making a handle or pointer refer to the same or another object on the CLI heap.

rvalue — This has the same meaning as that defined in the C++ Standard (§3.10).

tracking — The act of keeping track of the location of an object that resides on the CLI heap; this is necessary because such objects can move during their lifetime (unlike objects on the native heap, which never move). Tracking is maintained by the Virtual Execution System during garbage collection. Tracking is an inherent property of handles and tracking references.

type, boxed — See "type, value class, boxed".

type, class, any — Any CLI or native class type.

type, class, CLI class — A ref class type, a value class type, or an interface class type.

type, class, interface — A type that declares a set of virtual members that an implementing class shall define. An interface class type is a CLI type.

type, class, native — An ordinary Standard C++ class (declared using `class`, `struct`, or `union`).

type, class, ref — A type that can contain fields, function members, and nested types. A ref class type is a CLI type.

type, class, value — A type that can contain fields, function members, and nested types. Instances of a value class type are values. Since they directly contain their data, no heap allocation is necessary. A value class type is a CLI type.

type, class, value, boxed — A boxed value class is an instance of a value class on the CLI heap. For a value class `V`, a boxed value class is always of the form `V^`.

type, class, value, simple — The subset of value class types that can be embedded in a native class type and allocated with the `new` operator.

type, fundamental — The arithmetic types as defined by the C++ Standard (§3.9.1), and that each have a corresponding value class type provided by the implementation. (These include `bool`, `char`, and `wchar_t`, but exclude enumerations.)

type, handle — Longhand for "handle".

type, pointer, native — The pointer types as defined by the C++ Standard (§8.3.1). (Unlike a handle, a native pointer doesn't track, since objects on the native heap never move.)

type, reference, native — The reference types as defined by the C++ Standard (§8.3.2).

type, reference, tracking — A reference that can keep track of an object on the CLI heap when that object is moved by the garbage collector. For any type `T`, the declaration `T% r` declares a tracking reference `r` to type `T`. (See also "handle".)

unboxing — An explicit conversion from type `System::Object^` to any value class type, from type `System::ValueType^` to any value class type, from `V^` (the boxed form of a value class type) to `V` (the value class type), or from any interface class type handle to any value class type that implements that interface class. (See also "boxing".)

Virtual Execution System (VES) — This system implements and enforces the Common Type System (CTS) model. The VES is responsible for loading and running programs written for the CLI. It provides the services needed to execute CIL and data, using the metadata to connect separately generated modules together at runtime. For example, given an address inside the code for a function, it must be able to locate the metadata describing that function. It must also be able to walk the stack, handle exceptions, and store and retrieve security information. The VES is also known as the "Execution Engine".

5. Notational conventions

Various pieces of text from the C++ Standard appear verbatim in this standard. The C++ Standard is augmented by this C++/CLI Standard, with additions indicated by underlining, and deletions indicated using strike-through. For example:

> The rules for operators remain largely unchanged from Standard C++; however, the following rule in Standard C++ (§13.5/6) is augmented to allow static member functions:
>
> > A static member or a non-member operator function shall ~~either be a non-static member function or be a non-member function and~~ have at least one parameter whose type is a class, a reference to a class, a handle to a class, an enumeration, a reference to an enumeration, or a handle to an enumeration.

Unless otherwise noted, the following names are used as shorthand to refer to a type of their corresponding kind:

- `I` for interface class
- `N` for native type
- `R` for ref class
- `S` for simple value class
- `V` for value class

The CLI has its own set of naming conventions, some of which differ from established C++ programming practice. The CLI conventions have been used throughout this Standard; see Annex E.

Many source code examples use facilities provided by the CLI namespace `System`; however, that namespace is not explicitly referenced. Instead, there is an implied `using namespace System;` at the beginning of each of those examples. Similarly, examples using `cout` also assume that the `iostream` header has been included and there is an implied `using namespace std;` at the beginning of each of those examples.

In a number of examples, C++/CLI source code is shown with corresponding metadata. For expository purposes, a specific mapping between primitive C++ types and metadata types is assumed; however, that mapping need not be used by a conforming implementation. For example, type `int` is shown to map to `System::Int32` (which, in metadata, is referred to as `int32`). In the examples, C++/CLI source code is written in a constant-width font, and the corresponding metadata it written in the same font, but with a grey-shaded background. For example,

```
public ref struct D : B {
   ref class R { … };
};
```

```
.class public auto ansi D extends B {
  .class auto ansi nested public R extends [mscorlib]System.Object { … }
}
```

6. Acronyms and abbreviations

This clause is informative

The following acronyms and abbreviations are used throughout this Standard:

IEC — the International Electrotechnical Commission

IEEE — the Institute of Electrical and Electronics Engineers

ISO — the International Organization for Standardization

The following terms are defined in the CLI standard.

BCL — Base Class Library, which provides types to represent the built-in data types of the CLI, simple file access, custom attributes, security attributes, string manipulation, formatting, streams, and collections.

CIL — Common Intermediate Language

CLI — Common Language Infrastructure

CLS — Common Language Specification

CTS — Common Type System

VES — Virtual Execution System

End of informative text

7. General description

This Standard is intended for implementers, academics, and application programmers. As such, it contains a considerable amount of explanatory material that, strictly speaking, is not necessary in a formal language specification.

This standard is divided into the following subdivisions:

1. Front matter (clauses 1–7);
2. Language overview (clause 8);
3. The language syntax, constraints, semantics, and library (clauses 9–32);
4. Metadata generation (clauses 33–34);
5. Annexes

Examples are provided to illustrate possible forms of the constructions described. References are used to refer to related clauses. Notes are provided to give advice or guidance to implementers or programmers. Rational provides explantory material as to why something is or is not in this standard. Annexes provide additional information and summarize the information contained in this Standard.

Clauses 1–5, 7, and 9–34 form a normative part of this standard; Introduction, clauses 6 and 8, annexes, notes, examples, rationale, and the index, are informative.

Except for whole clauses or annexes that are identified as being informative, informative text that is contained within normative text is indicated in the following ways:

1. [*Example:* code fragment, possibly with some narrative … *end example*]
2. [*Note:* narrative … *end note*]
3. [*Rationale:* narrative … *end rationale*]

8. Language overview

This clause is informative.

This specification is a superset of Standard C++. This clause describes the essential features of this specification. While later clauses describe rules and exceptions in detail, this clause strives for clarity and brevity at the expense of completeness. The intent is to provide the reader with an introduction to the language that will facilitate the writing of early programs and the reading of later clauses.

8.1 Getting started

The canonical “hello, world” program can be written as follows:

```
int main() {
   System::Console::WriteLine("hello, world");
}
```

The source code for a C++/CLI program is typically stored in one or more text files with a file extension of `.cpp`, as in `hello.cpp`. Using a command-line compiler (called `cl`, for example), such a program can be compiled with a command line like

```
cl hello.cpp
```

which produces an application named `hello.exe`. The output produced by this application when it is run is:

```
hello, world
```

where the `WriteLine` function automatically adds a terminating newline.

The CLI library is organized into a number of namespaces, the most commonly used being `System`. That namespace contains a ref class called `Console`, which provides a family of functions for performing console I/O. One of these functions is `WriteLine`, which when given a string, writes that string plus a trailing newline to the console. (Examples from this point on assume that the namespace `System` has been the subject of a *using-declaration*.)

8.2 Types

Value class types differ from handle types in that variables of value class types directly contain their data, whereas variables of the handle types store handles to objects. With handle types, it is possible for two variables to reference the same CLI object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value classes, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

The example

```
ref class Class1 {
public:
   int Value;
   Class1() {
      Value = 0;
   }
};

int main() {
   int val1 = 0;
   int val2 = val1;
   val2 = 123;
```

```
    Class1^ ref1 = gcnew Class1;
    Class1^ ref2 = ref1;
    ref2->Value = 123;

    Console::WriteLine("Values: {0}, {1}", val1, val2);
    Console::WriteLine("Refs: {0}, {1}", ref1->Value, ref2->Value);
}
```

shows this difference. The output produced is

```
Values: 0, 123
Refs: 123, 123
```

The assignment to the local variable `val1` does not affect the local variable `val2` because both local variables have primitive types (which are also value class types), and each local variable of a primitive type has its own storage. In contrast, the assignment `ref2->Value = 123;` affects the CLI object that both `ref1` and `ref2` reference.

The lines

```
Console::WriteLine("Values: {0}, {1}", val1, val2);
Console::WriteLine("Refs: {0}, {1}", ref1->Value, ref2->Value);
```

deserve further comment, as they demonstrate some of the string formatting behavior of `Console::WriteLine`, which, in fact, takes a variable number of arguments. The first argument is a string, which can contain numbered placeholders like `{0}` and `{1}`. Each placeholder refers to a trailing argument with `{0}` referring to the second argument, `{1}` referring to the third argument, and so on. Before the output is sent to the console, each placeholder is replaced with the formatted value of its corresponding argument.

Developers can define new value class types through enum and value class definitions.

The following code shows an example of each kind of type definition. Later clauses describe type definitions in detail.

```
public enum class Color {
   Red, Blue, Green
};

public value struct Point {
   int x, y;
};

public interface class IBase {
   void F();
};

public interface class IDerived : IBase {
   void G();
};

public ref class A {
protected:
   virtual void H() {
      Console::WriteLine("A.H");
   }
};

public ref class B : A, IDerived {
public:
   virtual void F() {
      Console::WriteLine("B::F, implementation of IBase::F");
   }

   virtual void G() {
      Console::WriteLine("B::G, implementation of IDerived::G");
   }
```

```
protected:
   virtual void H() override {
      Console::WriteLine("B::H, override of A::H");
   }
};
public delegate void MyDelegate();
```

Types like `Color`, `Point`, and `IBase` above, which are not defined inside other types (i.e., they are top-level types), can have a type visibility specifier of either `public` or `private`. The use of `public` in this context indicates that the type is visible outside its parent assembly. Conversely, `private` indicates that the type is not visible outside its parent assembly. The default visibility for a top-level type is `private`.

8.2.1 Fundamental types and the CLI

Each of the fundamental types has a corresponding value class type provided by the implementation; the correspondence is implementation-defined. For example, one implementation might specify that `int` has the corresponding type `System::Int32`, while another specifies it has the corresponding type `System::Int64`. Using the keyword name has the usual Standard C++ meaning, while the corresponding CLI name indicates a particular CLI platform type. [*Example*: `int` specifies the implementation-defined "natural" integer type, whereas `Int32` specifies an integer type that is exactly 32 bits on any CLI platform. *end example*]

The table below lists the fundamental types and their corresponding CLI-provided type *in one implementation*. **For consistency, the examples in this Standard use the values in this table without continually re-stating "implementation-defined".**

Type	**Description**	**Corresponding CLI Value class type**
`bool`	Boolean type; a bool value is either true or false	`System::Boolean`
`char`	8-bit signed/unsigned integral type	`System::SByte` (with modopt for IsSignUnspecifiedByte)
`signed char`	8-bit signed integral type	`System::SByte`
`unsigned char`	8-bit unsigned integral type	`System::Byte`
`short`	16-bit signed integral type	`System::Int16`
`unsigned short`	16-bit unsigned integral type	`System::UInt16`
`int`	32-bit signed integral type	`System::Int32`
`unsigned int`	32-bit unsigned integral type	`System::UInt32`
`long`	32-bit signed integral type	`System::Int32` (with modopt IsLong)
`unsigned long`	32-bit unsigned integral type	`System::UInt32` (with modopt IsLong)
`long long int`	64-bit signed integral type	`System::Int64`
`unsigned long long int`	64-bit unsigned integral type	`System::Uint64`
`float`	Single-precision floating point type	`System::Single`
`double`	Double-precision floating point type	`System::Double`
`long double`	Extra-precision floating point type	`System::Double` (with modopt IsLong)
`wchar_t`	A 16-bit Unicode code unit	`System::Char`

Although they are not fundamental types, three other types provided in the CLI library are worth mentioning. They are:

- `System::Object`, which is the ultimate base type of all value and handle types
- `System::String`, a sequence of Unicode code units
- `System::Decimal`, a precise decimal type with at least 28 significant digits

C++/CLI has no keyword type names that can correspond to these.

8.2.2 Conversions

A number of new kinds of conversion have been defined. These include handle and parameter array conversion, among others.

8.2.3 CLI array types

A CLI array differs from a native array (C++ Standard §8.3.4) in that the former is allocated on the CLI heap, and can have a rank other than one. The rank determines the number of indices associated with each array element. The rank of a CLI array is also referred to as the ***dimensions*** of the CLI array. A CLI array with a rank of one is called a ***single-dimensional*** CLI array, and a CLI array with a rank greater than one is called a ***multi-dimensional*** CLI array.

Throughout this Standard, the term ***CLI array*** is used to mean an array in the CLI. A C++-style array is referred to as a ***native array*** or, more simply, ***array***, whenever the distinction is needed.

A CLI array type is declared using a built-in pseudo-template ref class having the following declaration:

```
namespace cli {
   template<typename T, int rank = 1>
   ref class array : System::Array {
   };
}
```

An example of using this pseudo-template is:

```
int main() {
   array<int>^ arr1D = gcnew array<int>(4) {10, 42, 30, 12};
   Console::Write("The {0} elements are:", arr1D->Length);
   for each (int i in arr1D) {
      Console::Write("{0,3}", i);
   }
   Console::WriteLine();
   array<int, 3>^ arr3D = gcnew array<int, 3>(10, 20, 30);
}
```

The output produced is:

```
The 4 elements are: 10 42 30 12
```

Handle `arr1D` can be made to refer to any one-dimensional array of `int`. It currently refers to one containing four `int` elements. The read-only property `Array::Length` contains the element count. Handle `arr3D` can be made to refer to any three-dimensional array of `int`. It currently refers to one of size 10x20x30, all of whose elements have the default value for `int`; that is, zero.

8.2.4 Type system unification

C++/CLI provides a “unified type system”. All value and handle types derive from the type `System::Object`. It is possible to call instance functions on any value, even values of fundamental types such as `int`. The example

```
int main() {
   Console::WriteLine((3).ToString());
}
```

calls the instance function `ToString` from type `System::Int32` on an integer literal, resulting in the string “`3`” being output. (Note that the seemingly redundant grouping parentheses around the literal 3, are not redundant; they are needed to get the tokens “3” and “.” instead of “3.”.)

The example

```
int main() {
   int i = 123;
   Object^ o = i;                // boxing
   int j = safe_cast<int>(o);    // unboxing
}
```

is more interesting. An `int` value can be converted to `System::Object^` and back again to `int`. This example shows both ***boxing*** and ***unboxing***. When a variable of a value class type needs to be converted to a handle type, a `System::Object` ***box*** is allocated to hold the value, and the value is copied into the box. ***Unboxing*** is just the opposite. When a `System::Object` box handle is cast back to its original value class type, the value is copied out of the box and into the appropriate storage location.

This type system unification provides value classes with the benefits of object-ness without introducing unnecessary overhead. For programs that don't need `int` values to act like CLI objects, `int` values are simply 32-bit values. For programs that need `int` values to behave like CLI objects, this capability is available on demand. This ability to treat instances of value class types as CLI objects bridges the gap between value classes and ref classes that exists in most languages. For example, a `Stack` class can provide `Push` and `Pop` functions that take and return `Object^` values.

```
public ref class Stack {
public:
   Object^ Pop() { … }
   void Push(Object^ o) { … }
};
```

Because C++/CLI has a unified type system, the `Stack` class can be used with elements of any type, including value class types like `int`.

8.2.5 Pointers, handles, and null

Standard C++ supports pointer types and null pointer constants. C++/CLI adds handle types and null values. To help integrate handles, and to have a universal null, C++/CLI defines the keyword `nullptr`. This keyword represents a literal having the null type. `nullptr` is referred to as the ***null value constant***. (No instances of the null type can ever be created, and the only way to obtain a null value constant is via this keyword.)

The definition of ***null pointer constant*** (which Standard C++ requires to be a compile-time expression that evaluates to zero) is augmented to include `nullptr`. The null value constant can be implicitly converted to any pointer or handle type, in which case it becomes a ***null pointer value*** or ***null value***, respectively. This allows `nullptr` to be used in relational, equality, conditional, and assignment expressions, among others.

```
Object^ obj1 = nullptr; // handle obj1 has the null value
String^ str1 = nullptr; // handle str1 has the null value
if (obj1 == 0);         // false (0 is boxed, the two handles differ)
if (obj1 == 0L);        // false    "        "        "        "
if (obj1 == nullptr);   // true

char* pc1 = nullptr;    // pc1 is the null pointer value
if (pc1 == 0);          // true as zero is a null pointer value
if (pc1 == 0L);         // true     "        "        "
if (pc1 == nullptr);    // true as nullptr is a null pointer constant

int n1 = 0;
n1 = nullptr;           // error, no implicit conversion to int
if (n1 == 0);           // true, performs integer comparison
if (n1 == 0L);          //          "        "        "
if (n1 == nullptr);     // error, no implicit conversion to int

if (nullptr);           // error
if (nullptr == 0);      // error, no implicit conversion to int
if (nullptr == 0L);     //          "        "        "
nullptr = 0;            // error, nullptr is not an lvalue
nullptr + 2;            // error, nullptr can't take part in arithmetic
```

```
Object^ obj2 = 0;       // obj2 is a handle to a boxed zero
Object^ obj3 = 0L;      // obj3     "       "       "
String^ str2 = 0;       // error, no conversion from int to String^
String^ str3 = 0L;      //          "       "       "         "
char* pc2 = 0;          // pc2 is the null pointer value
char* pc3 = 0L;         // pc3      "       "       "

Object^ obj4 = expr ? nullptr : nullptr;  // obj4 is the null value
Object^ obj5 = expr ? 0 : nullptr;        // error, no composite type

char* pc4 = expr ? nullptr : nullptr;  // pc4 is the null pointer value
char* pc5 = expr ? 0 : nullptr;        // error, no composite type

int n2 = expr ? nullptr : nullptr;  // error, no implicit conversion to
int
int n3 = expr ? 0 : nullptr;        // error, no composite type

sizeof(nullptr);           // error, the null type has no size, per se
typeid(nullptr);           // error
throw nullptr;             // error

void f(Object^);           // 1
void f(String^);           // 2
void f(char*);             // 3
void f(int);               // 4
f(nullptr);                // error, ambiguous (1, 2, 3 possible)
f(0);                      // calls f(int)

void g(Object^, Object^);  // 1
void g(Object^, char*);    // 2
void g(Object^, int);      // 3
g(nullptr, nullptr);       // error, ambiguous (1, 2 possible)
g(nullptr, 0);             // calls g(Object^, int)
g(0, nullptr);             // error, ambiguous (1, 2 possible)

void h(Object^, int);
void h(char*, Object^);
h(nullptr, nullptr);       // calls h(char*, Object^);
h(nullptr, 2);             // calls h(Object^, int);

template<typename T> void k(T t);
k(0);                      // specializes k, T = int
k(nullptr);                // error, can't instantiate null type
k((Object^)nullptr);       // specializes k, T = Object^
k<int*>(nullptr);          // specializes k, T = int*
```

Since objects allocated on the native heap do not move, pointers and references to such objects need not track an object's location. However, objects on the CLI heap can move, so they require tracking. As such, native pointers and references are not sufficient for dealing with them. To track objects on the CLI heap, C++/CLI defines handles (using the punctuator ^) and tracking references (using the punctuator %).

```
N* pn = new N;    // allocate on native heap
N& rn = *pn;      // bind ordinary reference to native object

R^ hr = gcnew R;  // allocate on CLI heap
R% rr = *hr;      // bind tracking reference to gc-lvalue
```

In general, the punctuator % is to ^ as the punctuator & is to *.

Just as Standard C++ has a unary & operator, C++/CLI provides a unary % operator. While `&t` yields a `T*` or an `interior_ptr<T>` (see below), `%t` yields a `T^`.

Rvalues and lvalues continue to have the same meaning as with Standard C++, with the following rules applying:

- An entity declared with type `T*`, a native pointer to `T`, points to an lvalue.
- Applying unary * to an entity declared with type `T*`, dereferencing a `T*`, yields an lvalue.
- An entity declared with type `T&`, a native reference to `T`, is an lvalue.
- The expression &lvalue yields a `T*`.

- The expression %lvalue yields a `T^`.

A ***gc-lvalue*** is an expression that refers to an object that might be on the CLI heap, or to a value member contained within such an object. The following rules apply to gc-lvalues:

- Standard conversions exist from "cv-qualified lvalue of type `T`" to "cv-qualified gc-lvalue of type `T`," and from "cv-qualified gc-lvalue of type `T`" to "cv-qualified rvalue of type `T`."
- An entity declared with type `T^`, a handle to `T`, points to a gc-lvalue.
- Applying unary `*` to an entity declared with type `T^`, dereferencing a `T^`, yields a gc-lvalue.
- An entity declared with type `T%`, a tracking reference to `T`, is a gc-lvalue.
- The expression &gc-lvalue yields an `interior_ptr<T>` (see below).
- The expression %gc-lvalue yields a `T^`.

The garbage collector is permitted to move objects that reside on the CLI heap. In order for a pointer to refer correctly to such an object, the runtime needs to update that pointer to the object's new location. An interior pointer (which is defined using `interior_ptr`) is a pointer that is updated in this manner.

8.3 Parameters

A ***parameter array*** is a type-safe alternative to parameter lists that end with an ellipsis.

A parameter array is declared with a leading `...` punctuator, followed by a CLI array type. There can be only one parameter array for a given function, and it shall always be the last parameter specified. The type of a parameter array is always a single-dimensional CLI array type. A caller can either pass a single argument of this CLI array type, or any number of arguments of the element type of this CLI array type. For instance, the example

```
void F(... array<int>^ args) {
   Console::WriteLine("# of arguments: {0}", args->Length);
   for (int i = 0; i < args->Length; i++)
      Console::WriteLine("\targs[{0}] = {1}", i, args[i]);
}

int main() {
   F();
   F(1);
   F(1, 2);
   F(1, 2, 3);
   F(gcnew array<int> {1, 2, 3, 4});
}
```

shows a function `F` that takes a variable number of `int` arguments, and several invocations of this function. The output is:

```
# of arguments: 0
# of arguments: 1
   args[0] = 1
# of arguments: 2
   args[0] = 1
   args[1] = 2
# of arguments: 3
   args[0] = 1
   args[1] = 2
   args[2] = 3
# of arguments: 4
   args[0] = 1
   args[1] = 2
   args[2] = 3
   args[3] = 4
```

By declaring the parameter array to be a CLI array of type `System::Object^`, the parameters can be heterogeneous; for example:

```
void G(... array<Object^>^ args) { … }
G(10, “Hello”, 1.23, ‘X’);       // arguments 1, 3, and 4 are boxed
```

A number of examples presented in this Standard use the `WriteLine` function of the `Console` class. The argument substitution behavior of this function, as exhibited in the example

```
int a = 1, b = 2;
Console::WriteLine("a = {0}, b = {1}", a, b);
```

is accomplished using a parameter array. The `Console` class provides several overloaded versions of the `WriteLine` function to handle the common cases in which a small number of arguments are passed, and one general-purpose version that uses a parameter array, as follows:

```
namespace System {
   public ref class Object { … };
   public ref class String { … };
   public ref class Console {
   public:
      static void WriteLine(String^ s) { … }
      static void WriteLine(String^ s, Object^ a) { … }
      static void WriteLine(String^ s, Object^ a, Object^ b) { … }
      static void WriteLine(String^ s, Object^ a, Object^ b, Object^ c)
         { … }
      …
      static void WriteLine(String^ s, ... array<Object^>^ args) { … }
   };
}
```

The CLI library specification shows library functions using C# syntax, in which case, the C# keyword `params` indicates a parameter array. For example, the declaration of the final `WriteLine` function above is written in C#, as follows:

```
public static void WriteLine(string s, params object[] args)
```

8.4 Automatic memory management

The example

```
public ref class Stack {
public:
   Stack() {
      first = nullptr;
   }

   property bool IsEmpty {
      bool get() {
         return (first == nullptr);
      }
   }

   Object^ Pop() {
      if (first == nullptr)
         throw gcnew Exception("Can't Pop from an empty Stack.");
      else {
         Object^ temp = first->Value;
         first = first->Next;
         return temp;
      }
   }

   void Push(Object^ o) {
      first = gcnew Node(o, first);
   }
```

```
   ref struct Node {
      Node^ Next;
      Object^ Value;
      Node(Object^ value) {
         Next = nullptr;
         Value = value;
      }
      Node(Object^ value, Node^ next) {
         Next = next;
         Value = value;
      }
   };
private:
   Node^ first;
};
```

shows a `Stack` class implemented as a linked list of `Node` instances. Node instances are created in the `Push` function and are garbage-collected when no longer needed. A `Node` instance becomes eligible for garbage collection when it is no longer possible for any code to access it. For instance, when an item is removed from the `Stack`, the associated `Node` instance becomes eligible for garbage collection.

The example

```
int main() {
   Stack^ s = gcnew Stack;
   for (int i = 0; i < 10; i++)
      s->Push(i);
   s = nullptr;
}
```

shows code that uses the `Stack` class. A `Stack` is created and initialized with 10 elements, and then the handle to it is assigned the value `nullptr`. Once the variable `s` is assigned the null value, the `Stack` and the associated 10 `Node` instances become eligible for garbage collection. The garbage collector is permitted to clean up immediately, but is not required to do so.

The garbage collector underlying C++/CLI can work by moving objects on the CLI heap around in memory, but this motion is invisible to most C++/CLI developers. For developers who are generally content with automatic memory management, but sometimes need fine-grained control or that extra bit of performance, C++/CLI provides the ability to ***pin*** objects on the CLI heap, to prevent temporarily the garbage collector from moving them. For example,

```
void f(int* p) { *p = 100; }

int main() {
   array<int>^ arr = gcnew array<int>(100);
   pin_ptr<int> pinp = &arr[0];  // pin arr's location
   f(pinp);                      // change arr[0]'s value
}
```

8.5 Expressions

C++/CLI augments the C++ Standard with respect to operators. For example:

- The addition of delegates requires the use of the function-call operator to invoke the functions encapsulated by a delegate.
- A new use of `typeid` has been added. For example, `Int32::typeid` results in a handle to a CLI object of type `System::Type` that describes the CLI type `Int32`.
- The cast operators are augmented to accommodate handle types.
- The `safe_cast` operator has been added.
- The operator `gcnew` has been added. This allocates memory from the CLI heap.
- The binary + and − operators are augmented to accommodate delegate addition and removal, respectively.

- Simple assignment is augmented to accommodate properties and events as the left operand.
- Compound assignment operators are synthesized from the corresponding binary operator (§19.7.4).

8.6 Statements

A new statement, `for each`, has been added. This statement enumerates the elements of a collection, executing a block for each element of that collection. For example:

```
void display(array<int>^ args) {
   for each (int i in args)
      Console::WriteLine(i);
}
```

A type is said to be a ***collection type*** if it implements the `System::Collections::IEnumerable` interface or implements some ***collection pattern*** by meeting a number of criteria.

8.7 Delegates

Delegates enable scenarios that Standard C++ programmers typically address with function adapters from the Standard C++ Library.

A delegate definition implicitly defines a class that is derived from the class `System::Delegate`. A delegate instance encapsulates one or more functions in an ***invocation list***, each member of which is referred to as a ***callable entity***. For instance functions, a callable entity is an instance and a member function on that instance. For static functions or global- or namespace-scope functions, a callable entity is just a member, global-, or namespace-scope function, respectively. Given a delegate instance and an appropriate set of arguments, one can invoke all of that delegate instance's callable entities with that set of arguments.

Consider the following example:

```
delegate void MyFunction(int value);   // define a delegate type

public ref struct A {
   static void F(int i) { Console::WriteLine("F:{0}", i); }
};

public ref struct B {
   void G(int i) { Console::WriteLine("G:{0}", i); }
};
```

The static function `A::F` and the instance function `B::G` both have the same parameter types and return type as `MyFunction`, so they can be encapsulated by a delegate of that type. Note that even though both functions are public, their accessibility is irrelevant when considering their compatibility with `MyFunction`. Such functions can also be defined in the same or different classes, as the programmer sees fit.

```
int main() {
   MyFunction^ d;                    // create a delegate reference
   d = gcnew MyFunction(&A::F);      // invocation list is A::F
   d(10);

   B^ b = gcnew B;
   d += gcnew MyFunction(b, &B::G); // invocation list is A::F B::G
   d(20);

   d += gcnew MyFunction(&A::F);     // invocation list is A::F B::G A::F
   d(30);

   d -= gcnew MyFunction(b, &B::G); // invocation list is A::F A::F
   d(40);
}
```

```
F:10
F:20
G:20
F:30
G:30
F:30
F:40
F:40
```

The constructor for a delegate needs two arguments when it is bound to a non-static member function: the first is a handle to an instance of a ref class, and the second designates the non-static member function within that ref class's type, using the syntax of a pointer to member. The constructor for a delegate needs only one argument when it is bound to a static function, or a global- or namespace-scope function; the argument designates that function, using the syntax of a pointer to member or pointer to function, as appropriate.

The invocation lists of two compatible delegates can be combined via the `+=` operator, as shown. In addition, callable entities can be removed from an invocation list via the `-=` operator, as shown. However, an invocation list cannot be changed once it has been created. Specifically, these operators create new invocation lists.

Once a delegate instance has been initialized, it is possible to indirectly call the functions it encapsulates just as if they were called directly (in the same order in which they were added to the delegate's invocation list), except the delegate instance's name is used instead. The value (if any) returned by the delegate call is that returned by the final function in that delegate's invocation list. If a delegate instance is null and an attempt is made to call the "encapsulated" functions, an exception of type `NullReferenceException` results.

8.8 Native and ref classes

8.8.1 Literal fields

A ***literal field*** is a field that represents a compile-time constant rvalue. The value of a literal field is permitted to depend on the value of other literal fields within the same program as long as they have been previously defined. The example

```
ref class X {
   literal int A = 1;
public:
   literal int B = A + 1;
};

ref class Y {
public:
   literal double C = X::B * 5.6;
};
```

shows two classes that, between them, define three literal fields, two of which are public while the other is private.

Even though literal fields are accessed like static members, a literal field is not static and its definition neither requires nor allows the keyword `static`. Literal fields can be accessed through the class, as in

```
int main() {
   cout << "B = " << X::B << "\n";
   cout << "C = " << Y::C << "\n";
}
```

which produces the following output:

```
   B = 2
   C = 11.2
```

Literal fields are only permitted in ref, value, and interface classes.

8.8.2 Initonly fields

The `initonly` identifier declares a field that is an lvalue only within the *ctor-initializer* and the body of an instance constructor, or within a static constructor, and thereafter is an rvalue. Such a field is called an ***initonly field***. For example:

```
public ref class Data {
   initonly static double coefficient1;
   initonly static double coefficient2;
   static Data() {
      // read in the value of the coefficients from some source
      coefficient1 = …; // ok
      coefficient2 = …; // ok
   }
public:
   static void F() {
      coefficient1 = …; // error
      coefficient2 = …; // error
   }
};
```

Assignments to an initonly field can only occur as part of its definition, or in an instance constructor or static constructor in the same class. (A static initonly field can be assigned to in a static constructor, and a non-static initonly field can be assigned to in an instance constructor.)

Initonly fields are only permitted in ref and value classes.

8.8.3 Functions

Member functions in CLI class types are defined and used just as in Standard C++. However, C++/CLI does have some differences in this regard. For example:

- The `const` and `volatile` qualifiers are not permitted on instance member functions.
- The function modifier `override` and override specifiers provide the ability to indicate explicit overriding and named overriding (§8.8.10.1).
- Marking a virtual member function as `sealed` prohibits that function from being overridden in a derived class.
- The function modifier `abstract` provides an alternate way to declare an abstract function.
- The function modifier `new` allows the function to which it applies to ***hide*** the base class function of the same name, parameter-type-list, and cv-qualification. Such a hiding function does not override any base class function, even if the hiding function is declared `virtual`.
- Type-safe variable-length argument lists are supported via parameter arrays.

8.8.4 Properties

A ***property*** is a member that behaves as if it were a field. There are two kinds of properties: scalar and indexed. A ***scalar property*** enables field-like access to a class or CLI object. Examples of scalar properties include the length of a string, the size of a font, the caption of a window, and the name of a customer. An ***indexed property*** enables array-like access to a CLI object. An example of an index property is a bit-array class.

Properties are an evolutionary extension of fields—both are named members with associated types, and the syntax for accessing scalar fields and scalar properties is the same, as is that for accessing CLI arrays and indexed properties. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessor functions*** that specify the statements to be executed when their values are read or written.

Properties are defined with property definitions. The first part of a property definition looks quite similar to a field definition. The second part includes a get accessor function and/or a set accessor function. Properties that can be both read and written include both get and set accessor functions. In the example below, the `Point` class defines two read-write properties, `X` and `Y`.

```
public value class Point {
   int Xor;
   int Yor;

public:
   property int X {
      int get()            { return Xor; }
      void set(int value)  { Xor = value; }
   }

   property int Y {
      int get()            { return Yor; }
      void set(int value)  { Yor = value; }
   }

   Point(int x, int y) {
      Move(x, y);
   }

   void Move(int x, int y) {        // absolute move
      X = x;
      Y = y;
   }

   void Translate(int x, int y) {   // relative move
      X += x;
      Y += y;
   }
   …
};
```

The get accessor function is called when the property's value is read; the set accessor function is called when the property's value is written.

The definition of properties is relatively straightforward, but the real value of properties is seen when they are used. For example, the `X` and `Y` properties can be read and written as though they were fields. In the example above, the properties are used to implement data hiding within the class itself. The following application code (directly and indirectly) also uses these properties:

```
Point p1;               // set to (0,0)
p1.X = 10;              // set to (10,0)
p1.Y = 5;               // set to (10,5)
p1.Move(5, 7);          // move to (5,7)
Point p2(9, 1);         // set to (9,1)
p2.Translate(-4, 12);   // move 4 left and 12 up, to (5,13)
```

For a ***trivial property*** declaration such as

```
property String^ Name;
```

the compiler automatically provides the default implementations of the accessor functions.

A ***default-indexed property*** allows array-like access directly on an instance. [*Note:* Other languages refer to default-indexed properties as "indexers". *end note*]

As an example, consider a `Stack` class. The designer of this class might want to expose array-like access so that it is possible to inspect or alter the items on the stack without performing unnecessary `Push` and `Pop` operations. That is, class `Stack` is implemented as a linked list, but it also provides the convenience of array access.

Default-indexed property definitions are similar to property definitions, with the main differences being that default-indexed properties are nameless and that they include indexing parameters. The indexing parameters are provided between square brackets. The example

```
public ref class Stack {
public:
   ref struct Node {
      Node^ Next;
      Object^ Value;
      Node(Object^ value) : Next(nullptr), Value(value) {}
      Node(Object^ value, Node^ next) {
         Next = next;
         Value = value;
      }
   };
private:
   Node^ first;
   Node^ GetNode(int index) {
      Node^ temp = first;
      while (index > 0) {
         temp = temp->Next;
         index--;
      }
      return temp;
   }
   bool ValidIndex(int index) { … }
public:
   property Object^ default[int] {     // default-indexed property
      Object^ get(int index) {
         if (!ValidIndex(index))
            throw gcnew Exception("Index out of range.");
         else
            return GetNode(index)->Value;
      }

      void set(int index, Object^ value) {
         if (!ValidIndex(index))
            throw gcnew Exception("Index out of range.");
         else
            GetNode(index)->Value = value;
      }
   }

   Object^ Pop() { … }
   void Push(Object^ o) { … }

   …
};
int main() {
   Stack^ s = gcnew Stack;

   s->Push(1);
   s->Push(2);
   s->Push(3);

   s[0] = 33;  // The top item now refers to 33 instead of 3
   s[1] = 22;  // The middle item now refers to 22 instead of 2
   s[2] = 11;  // The bottom item now refers to 11 instead of 1
}
```

shows a default-indexed property for the `Stack` class.

[*Note:* A more efficient implementation of Stack would make use of generics. *end note*]

8.8.5 Events

An ***event*** is a member that enables a class or CLI object to provide notifications. A class defines an event by providing an event declaration (which resembles a field declaration, though with an added `event` identifier) and an optional set of event accessor functions. The type of this declaration must be a handle to a delegate type (§8.7).

In the example

```
public delegate void EventHandler(Object^ sender, EventArgs^ e);

public ref class Button {
public:
   event EventHandler^ Click;
};
```

the `Button` class defines a `Click` event of type `EventHandler`. The `Click` member is only used on the left-hand side of the += and −= operators, or with the function-call operator (in which case, all the functions in the event's delegate list are called). The += operator adds a handler for the event, and the -= operator removes a handler for the event. The example

```
public ref class Form1 {
   Button^ Button1;
   void Button1_Click(Object^ sender, EventArgs^ e) {
      Console::WriteLine("Button1 was clicked!");
   }

public:
   Form1() {
      Button1 = gcnew Button;
   // Add Button1_Click as an event handler for Button1's Click event
      Button1->Click += gcnew EventHandler(this, &Form1::Button1_Click);
   }

   void Disconnect() {
      Button1->Click -= gcnew EventHandler(this, &Form1::Button1_Click);
   }
};
```

shows a class, `Form1`, that adds `Button1_Click` as an event handler for `Button1`'s `Click` event. In the `Disconnect` function, that event handler is removed.

Programmers who wants more control can get it by explicitly providing add and remove accessor functions. For example, the `Button` class could be rewritten as follows:

```
public ref class Button {
   EventHandler^ handler;
public:
   event EventHandler^ Click {
      void add(EventHandler^ e)    { handler += e; }
      void remove(EventHandler^ e) { handler -= e; }
   }
   …
};
```

This change has no effect on client code, but it allows the `Button` class more implementation flexibility. For example, the event handler for `Click` need not be represented by a field.

For a ***trivial event*** declaration such as

```
event EventHandler^ Click;
```

the compiler automatically provides the default implementations of the accessor functions.

8.8.6 Static operators

In addition to Standard C++ operator overloading, C++/CLI provides the ability to define operators that are static and/or take parameters of ^ type.

The following example shows part of an integer vector class:

```
public ref class IntVector {
   array<int>^ values;

public:
   property int Length {            // property
      int get() { return values->Length; }
   }
```

```
   property int default[int] {      // default-indexed property
      int get(int index) { return values[index]; }
      void set(int index, int value) { values[index] = value; }
   }

   IntVector(int length);

   IntVector(int length, int value);

// unary - (negation)
   static IntVector^ operator-(IntVector^ iv) {
      IntVector^ temp = gcnew IntVector(iv->Length);
      for (int i = 0; i < iv->Length; ++i) {
         temp[i] = -iv[i];
      }
      return temp;
   }

   static IntVector^ operator+(IntVector^ iv, int val) {
      IntVector^ temp = gcnew IntVector(iv->Length);
      for (int i = 0; i < iv->Length; ++i) {
         temp[i] = iv[i] + val;
      }
      return temp;
   }

   static IntVector^ operator+(int val, IntVector^ iv) {
      return iv + val;
   }
   …
};

int main() {
   IntVector^ iv1 = gcnew IntVector(4);      // 4 elements with value 0
   IntVector^ iv2 = gcnew IntVector(7, 2);   // 7 elements with value 2
   iv1 = -2 + iv2 + 5;
   iv2 = -iv1;
}
```

8.8.7 Instance constructors

Unlike Standard C++, C++/CLI supports static constructors (§8.8.9). As such, this specification refers to constructors as defined by the C++ Standard as being ***instance constructors***.

8.8.8 Destructors and finalizers

In Standard C++, cleanup code has traditionally been encapsulated by the destructor. While this approach provides a convenient and powerful way to abstract resources, resource leaks can occur if the destructor is not called. By having a garbage collector, C++/CLI provides a mechanism to write cleanup code that can be executed instead when an object is no longer referenced. As a result, a ref class can have two special member functions responsible for cleaning up resources held by an instance of that type: a destructor and a finalizer.

- **Destructor**: The destructor provides *deterministic cleanup* and ends the lifetime of the object. As in Standard C++, the destructor cleans up the bases and members of an object in the reverse order of the completion of their constructor. Within each ref class, in order, the destructor executes the user-written code, calls the destructors for each embedded member of the class, and calls the destructor for each base class. The main advantage of a destructor is that it is called at deterministic points in the program, which has the advantage of freeing resources earlier than if one waited for garbage collection.
- **Finalizer**: The finalizer provides *non-deterministic cleanup*. A finalizer is a “last-chance” function that is executed during garbage collection, typically on an object whose destructor was not executed. Finalizers are particularly useful to ensure resources that are represented by data members having value types (such as native pointers referring to allocation from the native heap) are cleaned up even if the destructor is not executed. The finalizer executes sometime

after the garbage collector determines there are no active references to the object. (There can be a performance penalty for having a finalizer.)

A ref class whose instances own resources should always have a destructor. A class that has a finalizer should always have a destructor as well, to enable deterministic cleanup and early resource release. However, a class that has a destructor need not necessarily have a finalizer.

```
ref struct R {
   ~R() { … }     // destructor, but no finalizer
};
```

A ref class whose instances have resources represented by value types (such as a pointer) should have a finalizer. (There may be a performance penalty for introducing a finalizer to a class that does not already have some finalizable ancestor class. As such, a well-designed class hierarchy will limit resources represented by value types to the leaves of the class hierarchy.) A ref class whose instances have no value types representing resources can still have a destructor, but should not have a finalizer.

```
ref struct R {
   ~R() { … }     // destructor
   !R() { … }     // finalizer
};
```

C++/CLI implements the destructor and finalizer semantics in any ref class `T` by using the ***CLI dispose pattern***, which makes use of five functions (`Dispose()`, `Dispose(bool)`, `Finalize()`, `__identifier("~T")()`, and `__identifier("!T")()`), all of whose definitions are generated by the compiler, as required. These cleanup mechanisms are hidden from the C++/CLI programmer. In C++/CLI, the proper way to do cleanup is to place all of the cleanup code in the destructor and finalizer, as follows:

- The finalizer should clean up any resources that are represented by value types.
- The destructor should do the maximal cleanup possible. To facilitate this, the programmer should call the finalizer from the destructor and write any other cleanup code in the destructor. A destructor can safely access the state of ref classes with references from the object, whereas a finalizer cannot.

For ref classes, both the finalizer and destructor must be written so they can be executed multiple times and on objects that have not been fully constructed.

8.8.9 Static constructors

A ***static constructor*** is a ref or value class static member function that implements the actions required to initialize the static members of a class, rather than the instance members of that class. Static constructors cannot have parameters, they must be private, and they cannot be called explicitly. The static constructor for a class is called automatically by the runtime. [*Note:* A static constructor is required to be private to prevent the static constructor from being invoked more than once. *end note*]

The example

```
public ref class Data {
private:
   initonly static double coefficient1;
   initonly static double coefficient2;
   static Data() {
      // read in the value of the coefficients from some source
      coefficient1 = …;
      coefficient2 = …;
   }
public:
   …
};
```

shows a `Data` class with a static constructor that initializes two initonly static fields.

8.8.10 Inheritance

When using ref classes, C++/CLI supports single inheritance of ref classes only. However, multiple inheritance of interfaces is permitted.

8.8.10.1 Function overriding

In Standard C++, given a derived class with a function having the same name, parameter-type-list, and cv-qualification as a virtual function in a base class, the derived class function always overrides the one in the base class, even if the derived class function is not declared `virtual`.

```
struct B {
   virtual void f();
   virtual void g();
};
struct D : B {
   virtual void f();    // D::f overrides B::f
   void g();            // D::g overrides B::g
};
```

We refer to this as ***implicit overriding***. (As the `virtual` specifier on `D::f` is optional, the presence of `virtual` there really isn't an indication of explicit overriding.) Since implicit overriding gets in the way of versioning (§8.13), implicit overriding must be diagnosed by a C++/CLI compiler.

C++/CLI supports two virtual function-overriding features not available in Standard C++. These features are available in ref class types. They are explicit overriding and named overriding.

Explicit overriding: In C++/CLI, it is possible to state that

1. A derived class function explicitly overrides a base class virtual function having the same name, parameter-type-list, and cv-qualification, by using the function modifier `override`, with the program being ill-formed if no such base class virtual function exists; and
2. A derived class function explicitly does not override a base class virtual function having the same name, parameter-type-list, and cv-qualification, by using the function modifier `new`.

```
ref struct B {
   virtual void F() {}
   virtual void G() {}
};
ref struct D : B {
   virtual void F() override {}  // D::F overrides B::F
   virtual void G() new {} // D::G doesn't override B::G, it hides it
};
```

`D::F` must be virtual, and must be marked as such. On the other hand, `D::G` doesn't have to be virtual, and if it isn't, it shouldn't be marked as such.

Named overriding: Instead of using the `override` modifier, we can achieve the same thing by using an *override-specifier*, which involves naming the function we are overriding. This approach also allows us to override a function having a different name, provided the parameter lists are the same.

```
ref struct B {
   virtual void F() {}
};
interface struct I {
   virtual void G();
};
ref struct D : B, I {
   virtual void X() = B::F, I::G {} // D::X overrides B::F and I::G
};
```

The use of `virtual` in all function declarations having an *override-specifier* is mandatory.

Explicit and named overriding can be combined, as follows:

```
ref struct B {
   virtual void F() {}
   virtual void G() {}
};
ref struct D : B {
   virtual void F() override = B::G {}
};
```

A function can only be overridden once in any given class. Therefore, if an implicit or explicit override does the same thing as a named override, the program is ill-formed.

```
ref struct B {
   virtual void F() {}
   virtual void G() {}
};
ref struct D : B {
   virtual void F() override = B::F {} // Error: B::F is overridden twice
   virtual void G() override {}        // B::G is overridden
   virtual void H() = B::G {}          // Error: B::G is overridden twice
};
```

[*Note:* If a base class is dependent on a template type parameter, a named override of a virtual function from that base class does not happen until the point of instantiation. In the following

```
template<typename T>
ref struct R : T {
   virtual void F() = T::G {}
};
```

`T::G` is a dependent name. *end note*]

8.9 Value classes

Value classes are similar to ref classes in that the former represent data structures that can contain fields and function members. However, unlike ref classes, value classes do not require heap allocation. A variable of a value class directly contains the data of the value class, whereas a variable of a ref class contains a handle to the data.

Value classes are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of value classes. Key to these data structures is that they have few fields, they do not require the use of inheritance or referential identity, and they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

The primitive types—such as `int`, `double`, and `bool`—are, in fact, all value class types. It is possible to use value class types and operator overloading to implement new "primitive" types.

```
value struct Point {
   int x, y;
   Point(int x, int y) {
      this->x = x;
      this->y = y;
   }
};
```

8.10 Interfaces

An interface defines a contract. A class that implements an interface must adhere to its contract by implementing all of the functions, properties, and events that interface declares.

The example

```
delegate void EventHandler(Object^ sender, EventArgs^ e);
```

```
interface class IExample {
   void F(int value);
   property bool P { bool get(); }
   property double default[int] {
      double get(int);
      void set(int, double);
   }
   event EventHandler^ E;
};
```

shows an interface that contains a function `F`, a read-only scalar property P, a default-indexed property, and an event `E`, all of which are implicitly public.

Interfaces are implemented using inheritance syntax.

```
interface class I1 { void F(); }; // F is implicitly virtual abstract
ref struct R1 : I1 { virtual void F() { /* implement I1::F */ } };
```

An interface can require implementation of one or more other interfaces. For example

```
interface class IControl {
   void Paint();
};
interface class ITextBox : IControl {
   void SetText(String^ text);
};
interface class IListBox : IControl {
   void SetItems(array<String^>^ items);
};
interface class IComboBox : ITextBox, IListBox {};
```

A class that implements `IComboBox` must also implement `ITextBox`, `IListBox`, and `IControl`.

Classes can implement multiple interfaces. In the example

```
interface class IDataBound {
   void Bind(Binder^ b);
};
public ref class EditBox : Control, IControl, IDataBound {
public:
   virtual void Paint() { … }
   virtual void Bind(Binder^ b) { … }
};
```

the class `EditBox` derives from the ref class `Control` and implements both `IControl` and `IDataBound`.

In the previous example, interface functions were implicitly implemented. C++/CLI provides an alternative way of implementing these functions that allows the implementing class to avoid having these members be public. Interface functions can be explicitly implemented using the named overriding syntax shown in §8.8.10.1. For example, the `EditBox` class could instead be implemented by providing `IControl::Paint` and `IDataBound::Bind` functions.

```
public ref class EditBox : IControl, IDataBound {
private:
   virtual void Paint() = IControl::Paint { … }
   virtual void Bind(Binder^ b) = IDataBound::Bind { … }
};
```

Interface members implemented in this way are called ***explicit interface members*** because each member explicitly designates the interface member being implemented.

```
int main() {
   EditBox^ editbox = gcnew EditBox;
   editbox->Paint();       // error: Paint is private
   IControl^ control = editbox;
   control->Paint();       // calls EditBox's Paint implementation
}
```

8.11 Enums

Standard C++ already supports enumerated types. However, C++/CLI provides some interesting extensions to this facility. For example:

- An enum can be declared public or private, so its visibility outside its parent assembly can be controlled.
- The underlying type for an enum can be specified.
- An enum type and/or its enumerators can have attributes.
- A new syntax is available for defining enums that are strongly typed and thus do not have integral promotions.

8.12 Namespaces and assemblies

The programs presented so far have stood on their own except for dependence on a few system-provided classes such as `System::Console`. It is far more common, however, for real-world applications to consist of several different pieces, each compiled separately. For example, a corporate application might depend on several different components, including some developed internally and some purchased from independent software vendors.

Namespaces and ***assemblies*** enable this component-based system. Namespaces provide a logical organizational system. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system—a way of presenting program elements that are exposed to other programs.

Assemblies are used for physical packaging and deployment. An assembly can contain types, the executable code used to implement these types, and references to other assemblies.

To demonstrate the use of namespaces and assemblies, this subclause revisits the "hello, world" program presented earlier, and splits it into two pieces: a class library that contains a function that displays the greeting, and a console application that calls that function.

The class library will contain a single class named `DisplayMessage`. For example:

```
// DisplayHelloLibrary.cpp
namespace MyLibrary {
   public ref struct DisplayMessage {
      static void Display() {
         Console::WriteLine("hello, world");
      }
   };
}
```

The next step is to write a console application that uses the `DisplayMessage` class; for example:

```
// HelloApp.cpp
#using <DisplayHelloLibrary.dll>
int main() {
      MyLibrary::DisplayMessage::Display();
}
```

No headers need to be included when using CLI library classes and functions. Instead, library assemblies are referenced via `#using` directives, with the assembly name enclosed in <…>, as shown. The code written can be compiled into a class library containing the class `DisplayMessage` and an application containing the function `main`. The details of this compilation step might differ based on the compiler or tool being used. A command-line compiler might enable compilation of a class library and an application that uses that library with the following command-line invocations:

```
cl /LD DisplayHelloLibrary.cpp
cl HelloApp.cpp
```

which produce a class library named `DisplayHelloLibrary.dll` and an application named `HelloApp.exe`.

8.13 Versioning

Versioning is the process of evolution of a component over time in a compatible manner. A new version of a component is ***source-compatible*** with a previous version if code that depends on the previous version can, when recompiled, work with the new version. In contrast, a new version of a component is ***binary-compatible*** if an application that depended on the old version can, without recompilation, work with the new version.

Consider the situation of a base class author who ships a class named `Base`. In the first version, `Base` contains no function `F`. A component named `Derived` derives from `Base`, and introduces an `F`. This `Derived` class, along with the class `Base` on which it depends, is released to customers, who deploy to numerous clients and servers.

```
public ref struct Base {      // version 1
   …
};
public ref struct Derived : Base {
   virtual void F() {
      Console::WriteLine("Derived::F");
   }
};
```

So far, so good, but now the versioning trouble begins. The author of `Base` produces a new version, giving it its own function `F`.

```
public ref struct Base {      // version 2
   virtual void F() {         // added in version 2
      Console::WriteLine("Base::F");
   }
};
```

This new version of `Base` should be both source and binary compatible with the initial version. (If it weren't possible simply to add a function then a base class could never evolve.) Unfortunately, the new `F` in `Base` makes the meaning of `Derived`'s `F` unclear. Did `Derived` mean to override `Base`'s `F`? This seems unlikely, since when `Derived` was compiled, `Base` did not even have an `F`! Further, if `Derived`'s `F` does override `Base`'s F, then it must adhere to the contract specified by `Base`—a contract that was unspecified when `Derived` was written. In some cases, this is impossible. For example, `Base`'s `F` might require that overrides of it always call the base. `Derived`'s `F` could not possibly adhere to such a contract.

C++/CLI addresses this versioning problem by allowing developers to state their intent clearly. In the original code example, the code was clear, since `Base` did not even have an `F`. Clearly, `Derived`'s `F` is intended as a new function rather than an override of a base function, since no base function named `F` exists.

If `Base` adds an `F` and ships a new version, then the intent of a binary version of `Derived` is still clear—`Derived`'s `F` is semantically unrelated, and should not be treated as an override.

However, when `Derived` is recompiled, the meaning is unclear—the author of `Derived` might intend its `F` to override `Base`'s `F`, or to hide it. By default, the compiler makes `Derived`'s `F` override `Base`'s `F`. However, this course of action does not duplicate the semantics for the case in which `Derived` is not recompiled.

If `Derived`'s `F` is semantically unrelated to `Base`'s `F`, then `Derived`'s author can express this intent by using the function modifier `new` in the declaration of `F`.

```
public ref struct Base {            // version 2
   virtual void F() {               // added in version 2
      Console::WriteLine("Base::F");
   }
};
public ref struct Derived : Base {  // version 2a: new
   virtual void F() new {
      Console::WriteLine("Derived::F");
   }
};
```

On the other hand, `Derived`'s author might investigate further, and decide that `Derived`'s `F` should override `Base`'s `F`. This intent can be specified explicitly by using the function modifier `override`, as shown below.

```
public ref struct Base {            // version 2
   virtual void F() {               // added in version 2
      Console::WriteLine("Base::F");
   }
};

public ref struct Derived : Base {  // version 2b: override
   virtual void F() override {
      Base::F();
      Console::WriteLine("Derived::F");
   }
};
```

The author of `Derived` has one other option, and that is to change the name of `F`, thus completely avoiding the name collision. Although this change would break source and binary compatibility for `Derived`, the importance of this compatibility varies depending on the scenario. If `Derived` is not exposed to other programs, then changing the name of `F` is likely a good idea, as it would improve the readability of the program—there would no longer be any confusion about the meaning of `F`.

8.14 Attributes

Standard C++ has certain declarative elements. For example, the accessibility of a function in a class can be specified by declaring it `public`, `protected`, or `private`. C++/CLI generalizes this capability, so that programmers can invent new kinds of declarative information, attach this declarative information to various program entities, and retrieve this declarative information at run-time. Programs specify this additional declarative information by defining and using ***attributes***.

For instance, a framework might define a `HelpAttribute` attribute that can be placed on program elements such as classes and functions, enabling developers to provide a mapping from program elements to documentation for them. The example

```
[AttributeUsage(AttributeTargets::All)]
public ref class HelpAttribute : Attribute {
   String^ url;
public:
   HelpAttribute(String^ url) {
      this->url = url;
   }

   String^ Topic;

   property String^ Url {
      String^ get() { return url; }
   }
};
```

defines an attribute class named `HelpAttribute` that has one positional parameter (`String^ url`) and one named parameter (`String^ Topic`). Positional parameters are defined by the formal parameters for public instance constructors of the attribute class, and named parameters are defined by public non-static read-write fields and properties of the attribute class. For convenience, usage of an attribute name when applying an attribute is allowed to drop the `Attribute` suffix from the name.

The example

```
[Help("http://www.mycompany.com/…/Class1.htm")]
public ref class Class1 {
public:
   [Help("http://www.mycompany.com/…/Class1.htm", Topic = "F")]
   void F() {}
};
```

shows several uses of the attribute `Help`.

Attribute information for a given program element can be retrieved at run-time by using reflection support. The example

```
int main() {
   Type^ type = Class1::typeid;
   array<Object^>^ arr =
      type->GetCustomAttributes(HelpAttribute::typeid, true);
   if (arr->Length == 0)
      Console::WriteLine("Class1 has no Help attribute.");
   else {
      HelpAttribute^ ha = (HelpAttribute^) arr[0];
      Console::WriteLine("Url = {0}, Topic = {1}", ha->Url, ha->Topic);
   }
}
```

checks to see if `Class1` has a `Help` attribute, and writes out the associated `Topic` and `Url` values if that attribute is present.

8.15 Generics

Generic types and functions are a set of features—collectively called ***generics***—defined by the CLI to allow parameterized types. Generics differ from templates in that generics are instantiated by the Virtual Execution System (VES) at runtime rather than by the compiler at compile-time. A generic definition must be a ref class, value class, interface class, delegate, or function.

8.15.1 Creating and consuming generics

Below, we create a `Stack` generic class definition where we specify a ***type parameter***, `ItemType`, using the same notation as with templates, except that the keyword `generic` is used instead of `template`. This type parameter acts as a placeholder until an actual type is specified at use.

```
generic<typename ItemType>
public ref class Stack {
   array<ItemType>^ items;
public:
   Stack(int size) {
      items = gcnew array<ItemType>(size);
   }

   void Push(ItemType data) { … }
   ItemType Pop() { … }
};
```

When we use the generic class definition `Stack`, we specify the actual type to be used by the generic class. In this case, we instruct the `Stack` to use an `int` type by specifying it as a ***type argument*** using the angle brackets after the name:

```
Stack<int>^ s = gcnew Stack<int>(5);
```

In so doing, we have created a new ***constructed type***, `Stack<int>`, for which every `ItemType` inside the definition of `Stack` is replaced with the supplied type argument `int`.

If we wanted to store items other than an `int` into a `Stack`, we would have to create a different constructed type from `Stack`, specifying a new type argument. Suppose we had a simple `Customer` type and we wanted to use a `Stack` to store it. To do so, we simply use the `Customer` class as the type argument to `Stack` and easily reuse our code:

```
Stack<Customer^>^ s = gcnew Stack<Customer^>(10);
s->Push(gcnew Customer);
Customer^ c = s->Pop();
```

Of course, once we've created a `Stack` with a `Customer` type as its type argument, we are now limited to storing only `Customer` objects (or objects of a class derived from `Customer`). Like templates, generics provide strong typing.

Generic type definitions can have any number of type parameters. Suppose we created a simple `Dictionary` generic class definition that stored values alongside keys. We could define a generic version of a `Dictionary` by declaring two type parameters, as follows:

```
generic<typename KeyType, typename ElementType>
public ref class Dictionary {
public:
   void Add(KeyType key, ElementType val) { … }
   property ElementType default[KeyType] { // indexed property
      ElementType get(KeyType key) { … }
      void set(KeyType key, ElementType value) { … }
   }
};
```

When we use `Dictionary`, we need to supply two type arguments within the angle brackets. Then when we call the `Add` function or use the indexed property, the compiler checks that we supplied the right types:

```
Dictionary<String^, Customer^>^ dict
   = gcnew Dictionary<String^, Customer^>;
dict->Add("Peter", gcnew Customer);
Customer^ c = dict["Peter"];
```

8.15.2 Constraints

In many cases, we will want to do more than just store data based on a given type parameter. Often, we will also want to use members of the type parameter to execute statements within our generic type definition. For example, suppose in the `Add` function of our `Dictionary` we wanted to compare items using the `CompareTo` function of the supplied key, as follows:

```
generic<typename KeyType, typename ElementType>
public ref class Dictionary {
public:
   void Add(KeyType key, ElementType val) {
      …
      if (key->CompareTo(val) < 0) { … }  // compile-time error
      …
   }
};
```

Unfortunately, at compile-time the type parameter `KeyType` is, as expected, generic. As written, the compiler will assume that only the operations available to `System::Object`, such as calls to the function `ToString`, are available on the variable `key` of type `KeyType`. As a result, the compiler will issue a diagnostic because the `CompareTo` function would not be found. However, we can cast the `key` variable to a type that does contain a `CompareTo` function, such as an `IComparable` interface, allowing the program to compile:

```
generic<typename KeyType, typename ElementType>
public ref class Dictionary {
public:
   void Add(KeyType key, ElementType val) {
      …
      if (static_cast<IComparable^>(key)->CompareTo(val) < 0) { … }
      …
   }
};
```

However, if we now construct a type from `Dictionary` and supply a key type argument which does not implement `IComparable`, we will encounter a run-time error (in this case, a `System::InvalidCastException`). Since one of the objectives of generics is to provide strong typing and to reduce the need for casts, a more elegant solution is needed.

We can supply an optional list of ***constraints*** for each type parameter. A constraint indicates a requirement that a type must fulfill in order to be accepted as a type argument. (For example, it might have to implement a given interface or be derived from a given base class.) A constraint is declared using the word `where`, followed by a type parameter and colon (`:`), followed by a comma-separated list of class or interface types.

In order to satisfy our need to use the `CompareTo` function inside `Dictionary`, we can impose a constraint on `KeyType`, requiring any type passed as the first argument to `Dictionary` to implement `IComparable`, as follows:

```
generic<typename KeyType, typename ElementType>
   where KeyType : IComparable
public ref class Dictionary {
public:
   void Add(KeyType key, ElementType val) {
      …
      if (key->CompareTo(val) < 0) { … }
      …
   }
};
```

When compiled, this code will now be checked to ensure that each time we construct a `Dictionary` type we are passing a first type argument that implements `IComparable`. Further, we no longer have to explicitly cast variable `key` to an `IComparable` interface before calling the `CompareTo` function.

Constraints are most useful when they are used in the context of defining a framework, i.e., a collection of related classes, where it is advantageous to ensure that a number of types support some common signatures and/or base types. Constraints can be used to help define "generic algorithms" that plug together functionality provided by different types. This can also be achieved by subclassing and runtime polymorphism, but static, constrained polymorphism can, in many cases, result in more efficient code, more flexible specifications of generic algorithms, and more errors being caught at compile-time rather than run-time. However, constraints need to be used with care and taste. Types that do not implement the constraints will not easily be usable in conjunction with generic code.

For any given type parameter, we can specify any number of interfaces as constraints, but no more than one base class. Each constrained type parameter has a separate `where` clause. In the example below, the `KeyType` type parameter has two interface constraints, while the `ElementType` type parameter has one class constraint:

```
generic<typename KeyType, typename ElementType>
   where KeyType : IComparable, IEnumerable
   where ElementType : Customer
public ref class Dictionary {
public:
   void Add(KeyType key, ElementType val) {
      …
      if (key->CompareTo(val) < 0) { … }
      …
   }
};
```

8.15.3 Generic functions

In some cases, a type parameter is not needed for an entire class, but only when calling a particular function. Often, this occurs when creating a function that takes a generic type as a parameter. For example, when using the `Stack` described earlier, we might often find ourselves pushing multiple values in a row onto a stack, and decide to write a function to do so in a single call.

We do this by writing a ***generic function***. Like a generic class definition, a generic function is preceded by the keyword `generic` and a list of type parameters enclosed in angle brackets. As in a template function, the type parameters of a generic function can be used within the parameter list, return type, and body of the function. A generic `PushMultiple` function might look like this:

```
generic<typename StackType, typename ItemType>
where ItemType : StackType
void PushMultiple(Stack<StackType>^ s, ... array<ItemType>^ values) {
   for each (ItemType v in values) {
      s->Push(v);
   }
}
```

Using this generic function, we can now push multiple items onto a `Stack` of any kind. Furthermore, because a constraint exists, the compiler type checking will ensure that the pushed items have the correct type for the kind of `Stack` being used. When calling a generic function, we place type arguments to the function in angle brackets; for example:

```
Stack<int>^ s = gcnew Stack<int>(5);
PushMultiple<int,int>(s, 1, 2, 3, 4);
```

The call to this function supplies the desired `StackType` and `ItemType` as type arguments to the function. In many cases, however, the compiler can deduce the correct type argument from the other arguments passed to the function, using a process called ***type deduction***. In the example above, since the first regular argument is of type `Stack<int>`, and the subsequent arguments are of type `int`, the compiler can reason that the type parameter must also be `int`. Thus, the generic `PushMultiple` function can be called without specifying the type parameter, as follows:

```
Stack<int>^ s = gcnew Stack<int>(5);
PushMultiple(s, 1, 2, 3, 4);
```

End of informative text.

9. Lexical structure

9.1 Tokens

9.1.1 Identifiers

Certain places in the Standard C++ grammar do not allow identifiers. However, C++/CLI allows a defined set of identifiers to exist in those places, with these identifiers having special meaning. [*Note:* Such identifiers are colloquially referred to as context-sensitive keywords; nonetheless, they are identifiers. *end note*] The identifiers that carry special meaning in certain contexts are:

```
abstract    delegate    event       finally     generic     in
initonly    internal    literal     override    property    sealed
where
```

When referred to in the grammar, these identifiers are used explicitly rather than using the *identifier* grammar production. Ensuring that the identifier is meaningful is a semantic check rather than a syntax check. An identifier is considered a keyword in a given context if and only if there is no valid parse if the token is taken as an identifier. That is, if it can be an identifier, it *is* an identifier.

Some naming patterns are reserved for function names in certain contexts (§19.2, §19.7.5).

When the token `generic` is found, it has special meaning if and only if it is not preceded by the token `::` or `typename`, and is followed by the token < and then either of the keywords `class` or `typename`. [*Note*: In rare cases, a valid Standard C++ program could contain the token sequence `generic` followed by < followed by `class`, where `generic` should be interpreted as a type name. For example:

```
template<typename T> struct generic {
  typedef int I;
};

class X {};
generic<class X> x1;
generic<class X()> x2;
```

In such cases, use `typename` to indicate that the occurrence of `generic` is a type name:

```
typename generic<class X> x1;
typename generic<class X()> x2;
```

or, in these particular cases, an alternative would be to remove the keyword `class` (that is, to not use the *elaborated-type-specifier*), for example:

```
generic<X> x1;
generic<X()> x2;
```

end note]

The grammar productions for *elaborated-type-specifier* (C++ Standard §7.1.5.3, §14.6, and §A.6) that mention `typename` are augmented as follows, to make *nested-name-specifier* optional in the first of the two applicable productions:

elaborated-type-specifier:
- *attributes*$_{opt}$ *class-key* `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
- *attributes*$_{opt}$ *class-key* `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *template*$_{opt}$ *template-id*
- *attributes*$_{opt}$ *enum-key* `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
- *attributes*$_{opt}$ `typename` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
- *attributes*$_{opt}$ `typename` `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id*

[*Note*: Revision of the C++ Standard is currently underway, and changes proposed in that revision alter this production. *end note*]

attributes is described in §29.

The C++ Standard (§14.6/3) is augmented, as follows:

> A*n* ~~*qualified-id*~~*identifier* that refers to a type and in which the *nested-name-specifier* depends on a *template-parameter* (14.6.2) shall be prefixed by the keyword `typename` to indicate that the ~~*qualified-id*~~*identifier* denotes a type, forming an *elaborated-type-specifier* (7.1.5.3).

and §14.6/5 is deleted:

> ~~The keyword typename shall only be used in template declarations and definitions, including in the return type of a function template or member function template, in the return type for the definition of a member function of a class template or of a class nested within a class template, and in the type-specifier for the definition of a static member of a class template or of a class nested within a class template. The keyword typename shall be applied only to qualified names, but those names need not be dependent. The keyword typename shall be used only in contexts in which dependent names can be used. This includes template declarations and definitions but excludes explicit specialization declarations and explicit instantiation declarations. The keyword typename is not permitted in a base-specifier or in a mem-initializer; in these contexts a qualified-id that depends on a template-parameter (14.6.2) is implicitly assumed to be a type name.~~

[*Note*: The presence of `typename` lets the programmer disambiguate otherwise ambiguous cases such as the token sequence `property :: X x;`. The declaration `property :: X x;` declares a member variable named `x` of type `property::X`, as it does in Standard C++. The token sequence `property typename :: X x;` declares a property named `x` of type `::X`. *end note*]

When name lookup for any of `array`, `interior_ptr`, `pin_ptr`, or `safe_cast` fails to find the name, and the name is not followed by a left angle bracket (<), the name is interpreted as though it were qualified with `cli::` and the lookup succeeds, finding the name in namespace `::cli`.

When name lookup for any of `array`, `interior_ptr`, `pin_ptr`, or `safe_cast` succeeds and finds the name in namespace `::cli`, the name is not a normal identifier, but has special meaning as described in this Standard.

Tokens that are not identifiers can be used as identifiers. This is achieved via `__identifier(`*T*`)`, where *T* shall be an *identifier*, a *keyword*, or a *string-literal*. The *string-literal* form is reserved for use by C++/CLI implementations. It is unspecified whether this replacement takes place before or after translation phase 4. [*Note*: Therefore, this construct should not be used in place of the first or only *identifier* in a `#define` preprocessing directive. *end note*] [*Example*:

```
__identifier(totalCost)
__identifier(delete)
__identifier("<Special Name #3>")
```

end example]

9.1.2 Keywords

The list of keywords in the C++ Standard (§2.11) is augmented by the following:

```
enum␣class        enum␣struct       for␣each          gcnew
interface␣class   interface␣struct  nullptr           ref␣class
ref␣struct        value␣class       value␣struct
```

The symbol ␣ is used in the grammar to signify that white-space appears within the keyword. Any white space that appears in the program text after translation phase 1 is permitted in the position signified by the ␣ symbol. It is unspecified whether white space generated by comments, documentation comments, and macro invocations is permitted in the position signified by the ␣ symbol. Following translation phase 4, a keyword with ␣ will be a single token. [*Note*: The ␣ symbol is only used in the grammar of the language.

Examples will include white-space as is required in a well-formed program. *end note*] [*Note*: Keywords that include the ⁜ symbol can be produced by macros, but are never considered to be macro names. *end note*]

Translation phase 4 in the C++ Standard (§2.1/4) is augmented as follows:

> Preprocessing directives are ~~executed~~parsed and stored. Then, in the translation unit and in each macro replacement-list, starting with the first token, each pair of adjacent tokens token1 and token2 is successively considered, and if token1⁜token2 is a keyword, then token1 and token2 are replaced with the single token token1⁜token2. ~~and~~ Then macro invocations are expanded. ...

In some places in the grammar, certain identifiers have special meaning, but are not keywords. [*Note:* For example, within a virtual function declaration, the identifiers `abstract` and `sealed` have special meaning. Ordinary user-defined identifiers are never permitted in these locations, so this use does not conflict with a use of these words as identifiers. For a complete list of these special identifiers, see §9.1.1. *end note*]

9.1.3 Literals

The grammar for *literal* in the C++ Standard (§2.13) is augmented as follows:

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal
null-literal

9.1.3.1 Integer literals

To accommodate the addition of the types `long long int` and `unsigned long long int`, the grammar for *integer-suffix* in the C++ Standard (§2.13.1) is augmented as follows:

integer-suffix:
unsigned-suffix *long-suffix*$_{opt}$
unsigned-suffix *long-long-suffix*$_{opt}$
long-suffix *unsigned-suffix*$_{opt}$
long-long-suffix *unsigned-suffix*$_{opt}$

long-long suffix: one of
`ll` `LL`

The C++ Standard (§2.13.1/2) is augmented as follows:

> The type of an integer literal depends on its form, value, and suffix. If it is decimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `long int`, `long long int`~~; if the value cannot be represented as a long int, the behavior is undefined~~. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, `unsigned long long int`. If it is suffixed by `u` or `U`, its type is the first of these types in which its value can be represented: `unsigned int`, `unsigned long int`, `unsigned long long int`. If it is decimal and is suffixed by `l` or `L`, its type is the first of these types in which its value can be represented: `long int`, ~~`unsigned long int`~~`long long int`. If it is octal or hexadecimal and is suffixed by `l` or `L`, its type is the first of these types in which its value can be represented: `long int`, `unsigned long int`, `long long int`, `unsigned long long int`. If it is suffixed by `ul`, `lu`, `uL`, `Lu`, `Ul`, `lU`, `UL`, or `LU`, its type is the first of these types in which its value can be represented: `unsigned long int`, `unsigned long long int`. If it is decimal and is suffixed by `ll` or `LL`, its type is `long long int`. If it is octal or hexadecimal and is suffixed by `ll` or `LL`, its type is the first of these types in which its value can be represented: `long long int`, `unsigned long long int`. If it is suffixed by both `u` or `U` and `ll` or `LL`, its type is `unsigned long long int`.

To accommodate the addition of extended integer types, the C++ Standard (§2.13.1/3) is augmented as follows:

> If an integer constant cannot be represented by any type in its list and an extended integer type can represent its value, then it has an extended integer type. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. A program is ill-formed if one of its translation units contains an integer literal that cannot be represented by any of the allowed types.

9.1.3.2 The null literal

> *null-literal:*
> `nullptr`

The *null-literal* is the keyword `nullptr`, whose type is the null type (§12.3.4). `nullptr` represents the ***null value constant*** and is unique. This literal is not an lvalue.

The null value constant can be converted to any handle type, with the result being a null handle. The null value constant can also be converted to any pointer type, with the result being a null pointer.

9.1.3.3 String literals

The C++ Standard (§2.13.4/1) is augmented as follows:

> … An ordinary string literal has type <narrow-string-literal-type>. This type cannot be named in the language, but it can be converted implicitly to either `System::String^` or array of *n* `const char`, as described in §14.2.5. ~~"array of *n* `const char`" and static storage duration (3.7), where *n* is the size of the string as defined below, and is initialized with the given characters.~~ … A wide string literal has type <wide-string-literal-type>. This type cannot be named in the language, but it can be converted implicitly to either `System::String^` or array of *n* `const wchar_t`, as described in §14.2.5. ~~"array of *n* `const wchar_t`" and has static storage duration, where *n* is the size of the string as defined below, and is initialized with the given characters.~~

9.1.4 Operators and punctuators

C++/CLI requires that template and generic constructs such as `List<List<int>>` be permitted, where >> is treated as two tokens instead of one. This requires augmentations to a number of places in the C++ Standard, as specified in this subclause and the subclauses §15.3, §30.1, and §30.2.

The C++ Standard (§2.1/1), translation phase 7, is augmented by adding the following text just prior to the existing note:

> [*Note:* The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (14.2). *end note*]

10. Basic concepts

10.1 Assemblies

The CLI defines an assembly as a configured set of loadable code modules and resources that together implement a unit of functionality. A C++/CLI program recognizes an assembly by the name of the file containing the assembly manifest. An assembly manifest describes all the constituent parts of the assembly such as the name of the assembly in metadata, other files that contribute to the assembly, and any hash codes that validate constituent parts.

An assembly can be an application or a library. An application has an application entry point, whereas a library does not.

10.2 Application entry point

In addition to the two definitions of the main function allowed in Standard C++ (see §3.6.1), C++/CLI allows the following definition:

```
int main(array<System::String^>^ args) { /* ... */ }
```

The value of `args` shall be a CLI array that represents the arguments to the program, where index 0 contains the first argument. If no arguments were passed to the program, `args` shall be a zero-length array; `args` shall never be null. The array passed to `main` is generated by the CLI runtime. [*Note*: Application entry points are described in §15.4.1.2 of the CLI Standard. *end note*]

10.3 Importing types from assemblies

Each type definition resides in some assembly, and an assembly can contain one or more types. The CLI Standard defines many types, each of which is defined in one of the three following assemblies: mscorlib.dll, System.dll, and System.Xml.dll. An application programmer can create any number of other assemblies, as needed.

A `#using` directive makes types from an assembly available in a source file; that is, it *imports* types from the metadata, and does not cause any types to be defined in the current translation unit. This directive has the following forms, which are equivalent:

```
#using < assembly-name >
#using " assembly-name "
```

[*Note*: Despite its appearance, `#using` is not a preprocessing directive. *end note*]

The types in assembly mscorlib.dll shall be implicitly imported by the compiler. [*Example*:

```
#using <mscorlib.dll>   // redundant
#using <System.dll>     // needed for Socket
#using <System.Xml.dll> // needed for XmlTextReader

int main() {
   System::Text::StringBuilder^ strBld;
   System::Net::Sockets::Socket^ soc;
   System::Xml::XmlTextReader^ xtr;
}
```

Each type has a namespace, a parent assembly, and a parent library; all three characteristics are separate and unrelated. For example, the type `Socket` is in the namespace `System::Net::Sockets`, the assembly System.dll, and the Networking library. *end example*]

For metadata details, see §34.1.1.

When a `#using` directive imports a type from an assembly, that type continues to belong to that assembly regardless of the number of other assemblies into which it is imported. On the other hand, when a `#include` preprocessing directive brings in a header containing a type definition, it brings in source code, which, when compiled, defines that type in the current translation unit.

When `#using` an assembly, if an imported type has a function with a signature that contains a modopt (§33.1) not defined by this Standard or one that has been used in a manner not defined by this Standard (for instance, using `IsSignUnspecifiedByte` (§33.1.5.7) on something other than a `System::Byte` or `System::SByte`), the following rules apply:

- If no other signature in the type is the same when ignoring the modopt, the compiler shall use the signature as if the modopt did not exist. Then if the function is virtual, any overriding function shall repeat the modopt.
- If when ignoring the modopt the function's signature is the same as another function's signature in the type, the compiler shall ignore the function with the unknown modopt, treating that function as if it did not exist.
- If there are two or more signatures with unknown modopts, and no signatures without modopts, all of the functions are ignored.

When `#using` an assembly, any value class type that has the `NativeCppClass` attribute (§33.2.1, 34.8), is treated as a native class, as described below. (If a type other than a value class has this attribute applied to it, the attribute is ignored and the type is treated as though the attribute had not been present.)

- A value class brought in from another assembly via `#using` is a forward declaration for that type.
- If a definition of the class is in source code, it is treated as the same class as that being brought it if the following criteria are met:
  - The source code definition has the same name as the encoding that came from `#using`.
  - The size of the source code definition is identical to the size in the encoding.
  - The visibility of the two need not be the same.

Being treated as "the same" means the following:

- Whenever the type from another assembly is used, the type defined in source code (in the current assembly) can be substituted. This is not a conversion.
- Whenever type information is needed for instructions such as call, the type used will match the function being called, but the type being supplied can be substituted by an object of the matching type in the current assembly.
- Whenever type information must be introduced in the current assembly (i.e., function parameter metadata), the type used shall be the type from the current assembly.
- The only exception is virtual overriding in a ref class. The signature of the virtual function shall match the original. Thus if the signature includes a native type, any function overriding it shall use the same type in its encoding.

All access to the native type using non-virtual functions shall be with functions from the current assembly. Member functions shall be private to each assembly.

When `#using` an assembly, if that assembly cannot be found or it is found but has an invalid format according to the CLI Standard, the compiler shall behave as if a corresponding `#error` directive was encountered.

10.4 Reserved names

There are certain functions that a programmer can never write in C++/CLI, but which may need to be imported from metadata created by translators of other languages. [*Example*: This can happen when a name

is reserved and cannot be written by the programmer; for example, `Finalize`, `Dispose`, or any of the operator function names. *end example*]

`#using` can import types with names that cannot be authored in C++/CLI. A C++/CLI programmer can use such a name in an expression when the reserved name does not have the meaning C++/CLI gives it. [*Example*: If a function named `Finalize` does not override the `Finalize` method from `System::Object`, a C++/CLI programmer can call the function `Finalize` without using the `!T` syntax (§19.13.2).

A second example involves the following C# class:

```
public class C : IDisposable {
  void IDisposable.Dispose() {}
  public void Dispose() {}
}
```

the function `C::Dispose` can be called from C++/CLI when `#using` that C# class because `C::Dispose` does not implement the `IDisposable::Dispose` function or override any function that does implement `IDisposable::Dispose`.

A third example is when an imported class has an implicit and explicit conversion operator that do the same thing. In this case, the compiler should just fall back to allowing the developer to write `op_Implicit` or `op_Explicit`. *end example*]

See also `__identifier` (§9.1.1).

10.5 Members

10.5.1 Value class members

The members of a value class are the members declared in that value class, and the members inherited from the value class's direct base class `System::ValueType` and the indirect base class `System::Object`.

The members of a fundamental type are the members of the corresponding value class type provided by the implementation (§12.1). [*Example*: The members of `signed char` are the members of the `System::SByte` value class. *end example*]

10.5.2 Delegate members

The members of a delegate are the members inherited from class `System::Delegate`, a public instance constructor, and the public methods `BeginInvoke`, `EndInvoke`, and `Invoke` (§34.14).

10.6 Member access

10.6.1 Declared accessibility

In the C++ Standard (§10), an *access-specifier* is used to define member access control. This grammar is augmented to accommodate the notion of assemblies, as follows:

access-specifier:
```
private
protected
public
internal
protected public
public protected
private protected
protected private
```

In the C++ Standard (§11/1), member access control for each *access-specifier* is defined. To accommodate the addition of assemblies, the list of definitions is augmented, as follows:

A member of a class can be

- `private`; that is, its name can be used only by members and friends of the class in which it is declared. This is referred to as private access.
- `protected`; that is, its name can be used only by members and friends of the class in which it is declared, and by members and friends of classes derived from this class (see 11.5). The parent assembly of derived classes does not affect protected access. This is referred to as family access.
- `public`; that is, its name can be used anywhere without access restriction. This is referred to as public access.
- `internal`; that is, its name can be used in its parent assembly. This is referred to as assembly access.
- `public protected` or `protected public`; that is, its name can be used in its parent assembly or by types derived from the containing class. This is referred to as family or assembly access.
- `private protected` or `protected private`; that is, its name can be used only by types derived from the containing class within its parent assembly. This is referred to as family and assembly access.

[*Note*: For *access-specifier*s containing two keywords, the more restrictive of the two applies outside the parent assembly while the less restrictive of the two applies within the parent assembly. *end note*]

An overriding name is allowed to have a different accessibility than the name it is overriding. An ordering is applied to distinguish between greater accessibility. Given the two accessibilities A and B, A has narrower access than B if A permits less access than A within the assembly and outside the assembly. A has wider access than B if A permits more access than A within the assembly and outside the assembly. Narrowing and widening of accessibilities implies a total ordering of accessibilities. For example, `protected` is wider than `private`, `protected` is narrower than `public`, `protected private` is narrower than `public protected`, and no ordering exists between `internal` and `protected`. [*Note:* In general, widening and narrowing accessibility is not CLS compliant. *end note*] When no ordering exsts between two accessibilities, one shall not be used to override the other.

When requirements are placed on wider or narrower accessibility, only the directly associated access specifier is considered. While accessiblity to a class member or type is determined by first checking accessibility of the enclosing entity, widening and narrowing rules do not consider the enclosing entity. [*Example*: The following code is valid.

```
public ref struct B {
  ref struct NB {
    virtual void F();
  };
};

private ref class D : B {
  ref class ND : B::NB {
  public:
    virtual void F() override;
  };
};
```

The overriding virtual function `F` in `ND` cannot have narrower accessibility than the virtual function `F` in `NB`. Since `NB::F` has public accessibility, `ND::F` must also have public accessibility. Both `D` and `ND` having private accessibility do not affect the narrowing rules. *end example*]

For metadata details, see §34.7.2.

10.7 Name lookup

The CLI (Partition I, §8.10.4) supports two different approaches to name lookup in base classes:

- If a derived member is marked *hide-by-name*, then functions in the base class with the same name are not visible in the derived class. This approach is referred to as ***hidebyname***.

- If a derived member is marked *hide-by-name-and-signature*, then functions in the base class with the same name and signature are not visible in the derived class. This approach is referred to as ***hidebysig***.

Implementation of the distinction between these two forms of hiding is provided entirely by source language compilers and the reflection library; it has no direct impact on the VES itself.

[*Note*: As in Standard C++, during lookup, whether the functions in a candidate set are static, virtual, or non-virtual, has no effect on overload resolution. *end note*]

The C++ Standard requires hidebyname lookup. As such, member functions of native classes use hidebyname lookup. [*Example*: Given the following program:

```
struct B {
   void F(int i) { … }
};
struct D : B {
   void F(String^ d) { … }
};
int main() {
   D d;
   d.F(100);
}
```

the function `F(String^)` is found, it's incompatible, and results in an error. *end example*]

On the other hand, member functions of ref classes, value classes, interface classes, and delegates use hidebysig lookup. [*Example*: Given the following program:

```
ref struct B {
   void F(int i) { … }
};
ref struct D : B {
   void F(String^ d) { … }
};
int main() {
   D d;
   d.F(100);
}
```

the function `F(int)` is called. *end example*]

If lookup for a name begins in a class, base interfaces are ignored.

If lookup for a name begins in an interface, when lookup proceeds to the bases of that interface, it shall continue searching for names in those interfaces.

The C++ Standard (§3.4/1) states:

> The access rules (clause 11) are considered only once name lookup and function overload resolution (if applicable) have succeeded.

In C++/CLI, that rule applies only to native classes. Otherwise, for CLI class types, inaccessible functions are not visible to name lookup. [*Note*: In Standard C++, a private name can hide names in a base class, whereas, in a CLI class type, a private name cannot hide names in a base class. *end note*]

[*Note*: In hidebyname, name lookup stops as soon as the name is found in a scope. In hidebysig, lookup continues unless the signature also matches. *end note*]

For qualified name lookup, lookup begins in the scope specified. If that scope uses hidebysig rules, then lookup uses hidebysig rules to find all names in the specified scope and other scopes. [*Example*: an expression such as *expr*`->R::F`, if `R` is a hidebysig class, lookup begins in `R`. Normal hidebysig rules apply, and thus a name set including names found in base classes of `R` is possible. *end example*]

Because hidebysig rules can create ambiguities between functions in a base class and a function in a derived class, the overload resolution rules are augmented to prefer functions in a derived class. [*Note*: Overload resolution is the same for candidate overload sets produced by hidebyname and hidebysig lookup. This can lead to ambiguity. *end note*]

In C++/CLI, functions in derived classes are preferred. To accomplish this, the C++ Standard (§13.3.3) is augmented, as follows:

> Given these definitions, a viable function `F1` is defined to be a better function than another viable function `F2` if for all arguments *i*, `ICS`*i*`(F1)` is not a worse conversion sequence than `ICS`*i*`(F2)`, and then
>
> — `F1` is a member of a more derived class than `F2` and neither `F1` nor `F2` are conversion functions, or if not that,
>
> — for some argument `j`, `ICS`*j*`(F1)` is a better conversion sequence than `ICS`*j*`(F2)`, or, if not that,
>
> …

[*Note*: With that rule, the program below will print "`float`". *end note*]

[*Example*:

```
ref struct B {
   void F(double) { Console::WriteLine("double"); }
};
ref struct D : B {
   void F(float) { Console::WriteLine("float"); }
};
int main() {
  D d;
  d.F(3.14);
}
```

The conversions from `(D^, double)` to `(B^, double)` and `(D^, float)` are equally ranked. Thus, with no additional rules the call would be ambiguous. *end example*]

If lookup in a class finds an entity that is not a function, lookup does not continue in the base classes. If lookup originated in a derived class, and the lookup set already contains a function, the entity in the base class is not included in the name set. (For the purpose of lookup, properties and events are treated as fields.) [*Example*:

```
ref struct A {
   void F(Object^) { Console::WriteLine("A::F"); }
};
ref struct B : A {
  int F;
};
ref struct C : B {
  void F(String^) { Console::WriteLine("C::F"); }
};
int main() {
  C c;
  c.F(4);      // error
}
```

No function `F` will be found because when lookup starts in `C`, it finds a function, then stops in `B` because a field with the same name exists. The same would happen if `B::F` were a property or event. *end example*]

A function scope is always hidebyname. As such, if lookup finds a name in function scope, it does not continue looking further. [*Example*:

```
ref struct R {
  void F(Object^) { Console::WriteLine("R::F(Object^)"); }

  void F() {
    extern void F(String^);
    F(4);      // error
    Console::WriteLine("R::F()");
  }
};

int main() {
  R r;
  r.F();
}

void F(String^) { Console::WriteLine("::F(String^)"); }
```

The program is ill-formed because the argument `4` cannot be converted to `String^`, which is the only viable function that lookup finds. *end example*]

A program that contains the definitions of two or more generic types with the same name and different arity (§31) in the same namespace, is ill-formed. However, a C++/CLI program can import such types from other assemblies with `#using`. When this happens, the ambiguity shall be resolved by counting the number of type arguments.

11. Preprocessor

11.1 Conditional inclusion

To accommodate the addition of the types `long long int` and `unsigned long long int`, and extended integer types, the C++ Standard (§16.1/4) is augmented, as follows:

> The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 using arithmetic that has at least the ranges specified in 18.2, except that ~~`int and unsigned int`~~ all signed and unsigned integer types act as if they have the same representation as, respectively, the largest signed integer type or unsigned integer type.

11.2 Predefined macro names

In addition to the macros specified in the C++ Standard (§16.8), the following macro name shall be defined by the implementation:

`__cplusplus_cli` The name `__cplusplus_cli` is defined to the value 200509L when compiling a C++/CLI translation unit. [*Note:* It is intended that future versions of this standard will replace the value of this macro with a greater value. *end note*]

The value of this predefined macro remains constant throughout the translation unit.

If this pre-defined macro name is the subject of a `#define` or a `#undef` preprocessing directive, the behavior is implementation-defined.

12. Types

All values in C++/CLI have a type. Types are grouped into seven categories as described in the following table.

Type Category	Type Subcategory
Native Class	POD Union
Ref Class	Boxed Value Type Delegate CLI Array
Value Type	Fundamental Type Enum Pointer Value Class
Interface	
Native Array	
Handle	
Reference	Native Reference Tracking Reference

Ref class types, value class types, and interface types are collectively known as ***CLI class types***.

The C++ Standard (§3.9/10) definition for ***scalar types*** is augmented, as follows:

> Arithmetic types (3.9.1), enumeration types, handle types, pointer types, and pointer to member types (3.9.2), and cv-qualified versions of these types (3.9.3) are collectively called scalar types.

The C++ Standard (§7.1.5) definition for *type-specifier* is augmented, as follows:

> *type-specifier:*
> *simple-type-specifier*
> *class-specifier*
> *enum-specifier*
> *elaborated-type-specifier*
> *cv-qualifier*
> *delegate-specifier*

To accommodate the addition of the types `long long int` and `unsigned long long int`, the C++ Standard (§7.1.5.2/Table 7) is augmented by the following rows:

Specifier(s)	Type
`long long`	`"signed long long int"`
`signed long long`	`"signed long long int"`
`long long int`	`"signed long long int"`
`signed long long int`	`"signed long long int"`
`unsigned long long`	`"unsigned long long int"`
`unsigned long long int`	`"unsigned long long int"`

12.1 Value types

Value types consist of the fundamental types, enums, pointers, and value classes. [*Note*: Standard C++ distinguishes between class types and non-class types; in C++/CLI, the fundamental types and enums have characteristics of both (see §12.1.1). All value types, with the exception of pointers, have the ability to be boxed through a boxing conversion (§14.2.6). *end note*]

Fundamental types are those that are "built-into" the language and have keywords associated with them. Enums are declared with the `enum`, `enum class`, or `enum struct` keywords. Pointers are declared using the asterisk in a declarator. Value classes are declared with the `value class` or `value struct` keywords.

12.1.1 Fundamental types

To accommodate the addition of the types `long long int` and `unsigned long long int`, and extended integer types, Standard C++ (§3.9.1) is augmented, as follows:

- §3.9.1/2: "There are ~~four~~five standard ***signed integer types***: "`signed char`", "`short int`", "`int`", ~~and~~ "`long int`", and "`long long int`". In this list, each type provides at least as much storage as those preceding it in the list. Plain `int`s have the natural size suggested by the architecture of the execution environment; the other signed integer types are provided to meet special needs. There may also be implementation-defined *extended signed integer types*. The standard and extended signed integer types are collectively called *signed integer types*."
- §3.9.1/3: "For each of the standard signed integer types, there exists a corresponding (but different) *standard* ***unsigned integer type***: "`unsigned char`", "`unsigned short int`", "`unsigned int`", ~~and~~ "`unsigned long int`", and "`unsigned long long int`", each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. Likewise, for each of the extended signed integer types there exists a corresponding *extended unsigned integer type* with the same amount of storage and alignment requirements. The standard and extended unsigned integer types are collectively called *unsigned integer types*. The range of nonnegative values of a *signed integer* type is a subrange of the corresponding *unsigned integer* type, and the value representation of each corresponding signed/unsigned type shall be the same. The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*, and the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*."
- §3.9.1, footnote 43): Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to ~~`int`, `unsigned int`, `long`, or `unsigned long`,~~ integral types as specified in 4.5."
- For all fundamental types (not just character types), all bits of the object representation participate in the value representation.
- An object of type `char` shall have exactly 8 bits.
- The value of an object having a signed integer type shall be stored using twos-complement representation.

The fundamental types map to corresponding value class types provided by the implementation, as follows:

- `signed char` maps to `System::SByte`.
- `unsigned char` maps to `System::Byte`.
- If a plain `char` is signed, `char` maps to `System::SByte`; otherwise, it maps to `System::Byte`.
- For all other fundamental types, the mapping is implementation-defined.

The representation of the `bool` value `false` shall be all-bits-zero.

In the C++ Standard, fundamental types are not considered class types; however, C++/CLI introduces class members to all fundamental types as every fundamental type shall map to a CLI class determined by the implementation. In C++/CLI, when a member selection operator is applied to an expression of fundamental type, or the scope resolution operator is applied to that fundamental type's keyword or typedef, in the scope of the expression containing the member selection operator or scope resolution operator, that fundamental type is treated as a class type. [*Note*: If a fundamental type is represented by more than one keyword, such as `unsigned int`, the scope resolution operator shall be applied to a typedef or the CLI class name to access static members. *end note*] As soon as the member selection operator or the scope resolution operator are used, C++/CLI uses the fundamental type's equivalent value class to resolve members. As member access and scope resolution are not allowed on fundamental types in the C++ Standard, all scenarios that distinguish between class and non-class types in the C++ Standard will always consider fundamental types as non-classes.

[*Example*: In the following example, the scope resolution operator applied to the keyword `int` results in looking for the name `Parse` in the associated CLI value class type. The member selection operator applied to the expression `x` with type `int` results in looking for the name `ToString` in the associated CLI value class type.

```
int x = int::Parse("42");
String^ s = x.ToString();
```

end example]

12.2 Class types

Ref class types, value class types, interface types, and delegate types shall not be declared at block scope.

12.2.1 Value classes

[*Note*: A value class is a data structure that can contain fields, function members, and nested types. Unlike other class types, value classes do not support user-defined destructors, finalizers, default constructors, copy constructors, or copy assignment operators. Value classes are designed to allow the CLI execution engine to efficiently copy value class objects.

All value class types implicitly inherit from the class `System::ValueType`, which, in turn, inherits from class `System::Object`. `System::ValueType` is not itself a value class type. Rather, it is a ref class type, from which all value class types are automatically derived.

Value classes are described in §22. *end note*]

12.2.2 Ref classes

[*Note*: A ref class defines a data structure that can contain fields, function members (functions, properties, events, operators, instance constructors, destructors, finalizers, and static constructors), and nested types. Ref classes support inheritance. Instances of ref classes are created using *new-expression*s (§15.4.6).

Ref classes are described in §21. *end note*]

12.2.3 Interface classes

[*Note*: An interface defines a contract. A ref or value class that implements an interface shall adhere to its contract. An interface can inherit from multiple base interfaces, and a ref or value class can implement multiple interfaces.

Interface classes are described in §25. *end note*]

12.2.4 Delegate types

[*Note*: A delegate is a data structure that refers to one or more functions, and for instance functions, it also refers to their corresponding instances.

Delegate types are described in §27. *end note*]

12.3 Declarator types

The C++ Standard (§8.3.5/3) is augmented, as follows:

> The resulting list of transformed parameter types and the presence or absence of the ellipsis is the function's *parameter-type-list*.

12.3.1 Raw types

A ***raw type*** is a class or fundamental type. [*Note*: This excludes "handle to" and "pointer to" types. *end note*]

12.3.2 Pointer types

It is possible to declare a pointer to a function that takes a parameter array (§18.4). [*Example:*

```
void F(double, ... array<int>^);
void (*p)(double, ... array<int>^) = &F;
```

end example]

A native pointer cannot point to a CLI heap-based object unless that object has been pinned (§12.3.7).

12.3.3 Handle types

For any CLI class type `T`, the declaration `T^ h` declares a handle `h` to type `T`, where the object to which `h` is capable of pointing resides on the CLI heap. A handle tracks, is rebindable, and can point to a whole CLI heap-based object only. [*Note:* In general, handles are to the gc heap as pointers are to the native heap. *end note*]

The default initial value of a handle shall be `nullptr`.

Objects of CLI class type are allocated on the CLI heap via `gcnew`, and such objects are referred to by handles. [*Example:*

```
R^ r1 = gcnew R;  // allocate an object on the CLI heap
R^ r2 = r1;       // handles r1 and r2 refer to the same object
```

end example]

If an object allocated using `gcnew` is never destroyed (using `delete` or by an explicit destructor call), that object's destructor will never be run; however, the garbage collector will reclaim the object's memory, and the object's finalizer (§19.13), if one exists, will be run. [*Example:*

```
{                          // allocate an object on the CLI heap
   R^ r3 = gcnew R;
}                          // the object will be garbage-collected and
                           // finalized, but its destructor will not be run
```

end example]

Unlike pointers, handles track; that is, a handle's value can change as the CLI heap-based object to which it refers is moved by the garbage collector. This has the following implications:

- A handle cannot be converted to and from `void*`. (A handle can, however, be converted to and from `Object^`.) [*Note:* There is no `void^`. *end note*]
- A handle cannot be converted to and from an integral type. (A handle cannot be hidden from the garbage collector.)
- Handles cannot be ordered.
- A handle can only point to a whole CLI heap-based object.

[*Example:*

```
R^ r4 = gcnew R;
Object^ o = r4;               // ok
R^ r5 = dynamic_cast<R^>(o);  // ok, r4 and r5 point to the same object
long l = reinterpret_cast<long>(r5); // error, can't convert to integer
R^ r6 = reinterpret_cast<R^>(l); // error, can't convert from integer
std::set<R^> s;               // error, R^'s can't be compared with less
```

end example]

All handles to the same CLI heap-based object compare equal, even if that object is moved by the garbage collector.

A handle can have any storage duration.

The representation of a handle with value `nullptr` shall be all-bits-zero.

12.3.4 Null type

The null type is a special type that exists solely to support the *null-literal*, `nullptr` (also referred to as the null value constant). No instances of this type can be created; the only way to obtain a value of this type is via the `nullptr` literal, whose type is the null type.

12.3.5 Reference types

A native reference can bind to any lvalue.

As an object on the CLI heap can be moved by the garbage collector, its location must be tracked. As such, a reference to such an object is called a ***tracking reference*** (%), and it can bind to any gc-lvalue. Whenever an object is definitively not on the CLI heap (as is the case if the object is an instance of a native class, a pinning pointer, or an interior pointer), the instance is an lvalue. [*Note*: As such, a native class does not need a copy assignment operator or copy constructor that takes gc-lvalues. An `N%` can be passed to these functions safely, since instances of native class types are never allocated on the CLI heap. An `N%` is an lvalue to begin with, so taking the address of an `N%` results in a native pointer, not an interior pointer. *end note*] [*Note:* Because there is a standard conversion from lvalue to gc-lvalue, a tracking reference can therefore bind to any gc-lvalue or lvalue. *end note*]

For any type `T`, the declaration `T% r` declares a tracking reference `r` to type `T`. [*Example:*

```
R^ h = gcnew R;   // allocate on CLI heap
R% r = *h;        // bind tracking reference to ref class object

void F(V% r);
F(*gcnew V);      // bind tracking reference to value class object
```

end example]

A tracking reference can refer to an instance of a ref class type, a cv-qualified value class type, a cv-qualified handle type, a cv-qualified native class type, or a cv-qualified native pointer. A program containing tracking references that refer to other types is ill-formed.

Like a native reference, a tracking reference is not rebindable; once set, its value cannot be changed.

A program containing a tracking reference that has storage duration other than automatic is ill-formed. (This precludes having a tracking reference as a data member.) [*Note:* This limitation directly reflects that of the CLI, because, in general, tracking references are implemented in terms of CLI managed pointers. *end note*]

Given an instance `v` of a value type `V`, `v` cannot be used as the object of a reference initialization if the reference is to a base class of `V`. (That is, `v` cannot reference bind to `System::Object%`, to `System::ValueType%`, or to any reference to an interface that `V` implements.) [*Rationale*: The reason for this is that such a reference binding would require boxing, yet binding a reference to a boxed value rather than to the original value defeats the purpose of reference binding. *end rationale*]

For metadata details, see §34.2.1.

12.3.6 Interior pointers

The garbage collector is permitted to move objects that reside on the CLI heap. In order for a pointer to refer correctly to such an object, the runtime needs to update that pointer to the object's new location. An ***interior pointer*** (declared using `interior_ptr`) is a pointer that is updated in this manner.

For metadata details, see §34.2.2.

12.3.6.1 Definitions

The compiler processes an interior pointer as follows:

- The compiler performs a lookup in the current context for the name `interior_ptr`.
- If the name refers unambiguously to `::cli::interior_ptr`, or the name is not found, then the expression is processed by the compiler according to the following grammar, and interpreted according to the rules specified herein.

```
interior_ptr < type-id >
```

An interior pointer shall have an implicit or explicit `auto` *storage-class-specifier*. An interior pointer can be used as a parameter and return type.

An interior pointer shall not be a class member or a base class.

The default initial value for an interior pointer shall be `nullptr`.

12.3.6.2 Target type restrictions

In the expression `interior_ptr<T>`, the target type `T` shall be a cv-qualified value class type, a cv-qualified handle type, a cv-qualified native class type, or a cv-qualified native pointer. A program containing other target types is ill-formed. [*Example:*

```
interior_ptr<int> p1;                  // OK
interior_ptr<int*> p2 = nullptr;       // OK
interior_ptr<System::String> p3;       // error, String is a ref class
interior_ptr<System::String^> p4;      // OK; is a handle to ref class
interior_ptr<interior_ptr<int>> p5;    // error, not a native pointer
interior_ptr<int^> p6 = nullptr;       // OK
```

end example]

12.3.6.3 Operations

An interior pointer can be involved in the same set of operations as native pointers, as defined by the C++ Standard. [*Note:* This includes comparison and pointer arithmetic. *end note*]

12.3.6.4 Data access

An interior pointer exhibits the usual pointer semantics for data access:

- Operator `->` is used to access a member of a CLI heap-based object pointed to by an interior pointer;
- Operator `*` is used to dereference an interior pointer.

[*Example:*

```
value struct V {
   int data;
};

V v;
interior_ptr<V> pv = &v;
pv->data = 42;
interior_ptr<int> pi = &v.data;
assert(*pi == 42);
```

end example]

Taking the address of an interior pointer yields a native pointer.

Interior pointers can point to objects inside the CLI heap. As such, taking the address of an object pointed to by an interior pointer yields an interior pointer that cannot be converted to `T*`.

[*Example:*

```
value struct V {
   int data;
};

V v;
interior_ptr<V> pv = &v;
V** p = &pv;                // error
interior_ptr<V>* pi = &pv; // OK, pv is on the stack and so is an lvalue
int* p2 = &(pv->data);     // error
int* p3 = &(v.data);       // OK, v is on the stack, v.data is an lvalue
```

end example]

12.3.6.5 The this pointer

In the body of a non-static member-function of a value class `V`, `this` is an rvalue expression of type `interior_ptr<V>`, whose value is the address of the CLI heap-based object for which the function is called.

[*Example:*

```
value struct V {
   int data;
   void f();
};

void V::f() {
   interior_ptr<V> pv1 = this;   // OK
   V* pv2 = this;                // error
}
```

end example]

12.3.7 Pinning pointers

Ordinarily, the garbage collector is permitted to move objects that reside on the CLI heap. However, such movement can be blocked temporarily, on a per object basis. A ***pinning pointer*** (declared using `pin_ptr`) is a pointer that prevents the garbage collector from moving the CLI heap-based object to which that pointer points. This makes it possible for code not under the control of the runtime to manipulate memory within the bounds of the CLI heap without corrupting that heap.

Although a pinning pointer can be initialized from an interior pointer, the value of a pinning pointer is never changed by the runtime.

A pinning pointer can point to an object anywhere in memory; it need not point to an object on the CLI heap.

For metadata details, see §34.2.3.

12.3.7.1 Definitions

The compiler processes a pinning pointer as follows:

- The compiler performs a lookup in the current context for the name `pin_ptr`.
- If the name refers unambiguously to `::cli::pin_ptr`, or the name is not found, then the expression is processed by the compiler according to the following grammar, and interpreted according to the rules specified herein.

`pin_ptr < `*type-id*` >`

A pinning pointer is an interior pointer that is a handle to type *type-specifier*; it is a *type-id*.

A pinning pointer shall have an implicit or explicit `auto` *storage-class-specifier*. A pinning pointer shall not be used as a parameter type or return type.

[*Note:* As a pinning pointer is an interior pointer, the default initial value for a pinning pointer is `nullptr`. (§12.3.6.1) *end note*]

12.3.7.2 Target type restrictions

The target type restrictions for pinning pointers are the same as for interior pointers (§12.3.6.2).

12.3.7.3 Operations

The operations that can be formed on pinning pointers are the same as for interior pointers (§12.3.6.3) except that a pinning pointer cannot be the target of a cast.

12.3.7.4 Data access

With two exceptions, pinning pointers follow the same data access semantic as interior pointers (§12.3.6.4). Since a pinning pointer points to an unmovable object inside the CLI heap, `pin_ptr<T>` can be converted to `T*`. Dereferencing a pinning pointer yields an lvalue. [*Example:*

```
value struct V {
   int data;
   void f();
};

void V::f() {
   int* pi;
   interior_ptr<V> ipv = this;
   pi = &(ipv->data);            // error
   pin_ptr<V> ppv = this;
   pi = &(ppv->data);            // OK

   V* pv;
   pv = ipv;                     // error
   pv = ppv;                     // OK
}

V v;
pin_ptr<V> pv = &v;
V** p = &pv;                     // error
int* pi = &pv->data;             // OK
```

end example]

12.3.7.5 Duration of pinning

As soon as a pinning pointer is initialized or assigned the address of a CLI heap-based object, that object is guaranteed to remain at its location. If the pinning pointer is then made to point to another CLI heap-based object, that object is guaranteed to remain at its location, and the object previously pointed to is no longer pinned by that pointer, allowing it to be moved. If a pinning pointer is assigned the value `nullptr`, the object previously pointed to (if any) is no longer considered pinned

When the block in which a pinning pointer is defined exits, any CLI heap-based object pointed to by that pinning pointer is no longer considered pinned by that pinning pointer; however, it might still be pinned by another pinning pointer.

With the exception of the functionality provided by the class `System::Runtime::InteropServices::GCHandle`, if no pinning pointer points to a CLI heap-based object, it is not safe to assume that object is pinned.

[*Example:*

```
ref struct R {
   int data;
};
R^ r = gcnew R;
{
   pin_ptr<int> ppi = &r->data;  // object referenced by r is pinned
}
// ppi's parent block has exited, so object is free to move
```

end example]

12.3.8 Native arrays

A program that contains a native array of elements having CLI class type or handle type, is ill-formed. [*Note*: Allowing elements of such types would make the array type a mixed type (§23). *end note*]

A native array type is local to its parent assembly (i.e., it is `internal`), and that type is not verifiable. Thus, a virtual function taking a native array type as a parameter cannot be overridden from another assembly.

For metadata details, see §34.2.4.

12.4 Top-level type visibility

A non-nested class, interface, delegate, or enum definition can optionally specify the visibility of the class, interface, delegate, or enum:

top-level-visibility:
`public`
`private`

The `public` *top-level-visibility* specifier indicates that the non-nested class, interface, delegate, or enum is visible outside its parent assembly. Conversely, the `private` *top-level-visibility* specifier indicates that the class, interface, delegate, or enum is not visible outside its parent assembly. However, private types are visible within their parent assembly. The default visibility for a class, interface, delegate, or enum is `private`. [*Example:*

```
public class VisibleClass {};    // visible outside the assembly
private class InternalClass {};  // visible only within the assembly
```

end example]

Those class, interface, delegate, or enum definitions nested within another type definition have the accessibility specified within that type. The use of a *top-level-visibility* modifier on a nested type definition causes the program to be ill-formed.

13. Variables

This part of this clause is informative.

In Standard C++, the term *variable* is used to designate a named object (C++ Standard §3/4, "Basic concepts"):

> A *name* is a use of an identifier (2.10) that denotes an entity or *label* (6.6.4, 6.1). A *variable* is introduced by the declaration of an object. The variable's name denotes the object.

In Standard C++, the term *object* refers to a region of data storage. (C++ Standard §1.8/1, "The C++ object model "):

> The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is a region of storage. [*Note*: A function is not an object, regardless of whether or not it occupies storage in the way that objects do.]

The term *CLI object* refers to any instance of CLI class type. The term *native object* refers to an instance of a native class.

End of informative text.

13.1 gc-lvalues

In Standard C++, every expression is either an lvalue or rvalue. In C++/CLI, an expression can also be a ***gc-lvalue***, which refers to an object that might be tracked by the garbage collector. Except where noted below, expectations for lvalues and rvalues based on Standard C++, are unchanged. In C++/CLI, every expression is either an lvalue, a gc-lvalue, or rvalue.

Some built-in operators yield gc-lvalues. [*Example*: If `E` is an expression of type "handle to type", then `*E` is a gc-lvalue. As the function `int% f();` yields a gc-lvalue, the call `f()` is a gc-lvalue. *end example*]

Some operators produce results that depend on whether the operand is an lvalue or gc-lvalue. [*Example*: One such operator is unary `&`. *end example*]

The result of calling a function returning a tracking reference, is a gc-lvalue, unless the tracking reference refers to a native class.

Whenever an lvalue appears in a context where a gc-lvalue is expected, the lvalue is converted to a gc-lvalue. Likewise, whenever a gc-lvalue appears in a context where an rvalue is expected, the gc-lvalue is converted to an rvalue.

Reference initialization and temporaries shall have semantics that make allowance for gc-lvalues, as well as lvalues and rvalues.

Like an lvalue, a gc-lvalue can have any complete type, the `void` type, or an incomplete type.

Like with an lvalue, to modify an object, a gc-lvalue for that object shall be used.

A program that attempts to modify an object through a nonmodifiable gc-lvalue is ill-formed.

The list of restrictions in the C++ Standard (§3.10/15) for accessing the stored value of an object through an lvalue also applies to gc-lvalues.

13.1.1 Standard conversions

The C++ Standard (§4.1) is augmented by the following:

> Any lvalue can be converted to a gc-lvalue. A gc-lvalue can convert to an rvalue in exactly the same cases as a conversion from lvalue to an rvalue. A program that necessitates any other lvalue to gc-lvalue or gc-lvalue to rvalue conversion is ill-formed.

13.1.2 Expressions

The C++ Standard (§5/6) is augmented by the following:

> If an expression initially has the type "reference to `T`" (8.3.2, 8.5.3), the type is adjusted to "`T`" prior to any further analysis, the expression designates the object or function denoted by the reference, and the expression is an lvalue. If an expression initially has the type "tracking reference to T", the type is adjusted to "T" prior to any further analysis, the expression designates the object or function denoted by the reference, and the expression is a gc-lvalue.

In general, in any context in which this clause determines the result of an expression is an lvalue because the resulting entity is a function, variable, or data member, it is an lvalue only if the entity is a function, or it is a variable or data member that is not on the CLI heap. If the entity is a variable or data member that is, or could be, on the CLI heap, the result is a gc-lvalue. This applies to cases mentioned in the C++ Standard, §5.1/4, §5.1/7, and §5.1/8. An entity of an expression might not always be on the CLI heap, but it might be. [*Example*: A member function of a value class referring to a data member of that value class shall assume that the class is allocated on the CLI heap, and is, therefore, a gc-lvalue. *end example*]

The C++ Standard (§5.2.2/10) is augmented as follows:

> A function call is an lvalue if and only if the result type is a native reference. A function call is a gc-lvalue if and only if the result type is a tracking reference.

The C++ Standard (§5.2.5/4) is augmented as follows:

> — If `E2` is a member enumerator, and the type of `E2` is `T`, the expression `E1.E2` is~~not an lvalue~~ an rvalue. The type of `E1.E2` is `T`.

The following rules have been added to the requirements of the C++ Standard (§5.2.5/4):

> — If `E2` is a static data member of a ref class or value class, and the type of `E2` is `T`, then `E1.E2` is a gc-lvalue; the expression designates the named member of the class. The type of `E1.E2` is `T`.
>
> — If `E2` is a non-static data member, the expression designates the named member of the object designated by the first expression. If `E1` is a gc-lvalue, then `E1.E2` is a gc-lvalue.

The C++ Standard (§5.2.6/1) is augmented as follows:

> … The operand shall be a modifiable gc-lvalue. …

The C++ Standard (§5.3.1/1) is augmented as follows:

> The unary `*` operator performs indirection: the expression to which it is applied shall be a pointer or handle to an object type, or a pointer to a function type~~. and t~~The result of applying indirection to a pointer is an lvalue referring to the object or function to which the expression points. The result of applying indirection to a handle is a gc-lvalue referring to the object. If the type of the expression is "pointer to `T`," the type of the result is "`T`." If the type of the expression is "`T^`," the type of the result is "`T`." [*Note*: a pointer to an incomplete type (other than *cv* `void`) can be dereferenced. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue shall not be converted to an rvalue, see 4.1.]

The C++ Standard (§5.3.1/2) is augmented as follows:

> The result of the unary `&` operator is a pointer to its operand. The operand shall be an lvalue, gc-lvalue, or a *qualified-id*. If the operand is an lvalue, given the type of the expression is "T", the result is an rvalue and its type is "pointer to T." If the operand is a gc-lvalue, given the type of the expression is "T", the result is an rvalue and its type is "`interior_ptr` to T."~~In the first case, if the type of the expression is "T," the type of the result is "pointer to T."~~ In particular, the address of an object of type "*cv* T" is "pointer to *cv* T," with the same cv-qualifiers. For a *qualified-id*, if the

member is a static member of type "T", the type of the result is plain "pointer to T." If the member is a nonstatic member of class C of type T, the type of the result is "pointer to member of class C of type T."

The C++ Standard (§5.3.2/1) is augmented as follows:

… The operand shall be a modifiable gc-lvalue. …

The primary list in the C++ Standard (§5.16/3) is augmented by the following:

— If `E2` is a gc-lvalue, `E1` can be converted to match `E2` if `E1` can be implicitly converted to the type "tracking reference to `T2`", subject to the constraint that in the conversion the reference shall bind directly to `E1`.

The C++ Standard (§5.16/4) is augmented by the following:

If the second and third operands are lvalues and have the same type, the result is of that type and is an lvalue. If the second and third operands are gc-lvalues and have the same type, the result is of that type and is a gc-lvalue.

The C++ Standard (§5.17/1) is augmented as follows:

There are several assignment operators, all of which group right-to-left. All require a modifiable gc-lvalue or lvalue as their left operand, and the type of an assignment expression is that of its left operand. The result of the assignment operation is the value stored in the left operand after the assignment has taken place~~; the result is an lvalue~~. The result of an assignment operator is an lvalue if the left operand was an lvalue. Likewise, the result of an assignment operator is a gc-lvalue if the left operand was a gc-lvalue.

The C++ Standard (§5.18/1) is augmented by the following:

The type and value of the result are the type and value of the right operand; the result is an lvalue if its right operand is. The result is a gc-lvalue if its right operand is a gc-lvalue.

13.1.3 Reference initializers

The C++ Standard (§8.5.3) is augmented by the following:

A native reference cannot bind to a gc-lvalue. If a native reference is bound to an rvalue, a temporary of the initializer expression shall be created (as described in Standard C++ §8.5.3/5). The temporary shall be allocated in memory not under control of the CLI heap.

A tracking reference can bind to an lvalue or a gc-lvalue. Unlike native references, a tracking reference need not be const to bind to an rvalue. That is, int% r = 42; is well-formed. Binding of tracking references otherwise follows the same rules as native references.

A native reference expression is always considered an lvalue. A tracking reference expression is always considered a gc-lvalue, except when the tracking reference refers to a native class, in which case, it is an lvalue.

13.1.4 Temporary objects

The C++ Standard (§12.2) is augmented by the following:

A temporary object is an rvalue, which shall not be allocated on the native heap.

13.2 File-scope and namespace-scope variables

For metadata details, see §34.3.1.

13.3 Direct initialization

Direct initialization in the C++ Standard (§8.5) occurs in `new` expressions, `static_cast` expressions, functional notation type conversions, and base and member initializers. Direct initialization considers both

constructors and user-defined conversion functions. C++/CLI makes a distinction amongst these different forms of direct initialization for CLI class types and limits usage of constructors and conversion functions to specific cases.

- If the initialization is taking place in a `new` expression and the destination type is a CLI class type, only constructors of the destination type are considered. [*Note*: Such a *new* expression, will only use the `gcnew` form of the grammar. *end note*] The C++ Standard (§8.5/14) is augmented for this case to remove any reference to conversion functions.
- If the initialization is taking place in a `static_cast` expression and the destination type is a CLI class type, only conversion functions of both the source type and destination type are considered. The C++ Standard (§8.5/14) is augmented for this case to remove any reference to constructors.
- If the initialization is taking place in a functional notation type conversion and the destination type is a CLI class type, only constructors of the destination type are considered. The C++ Standard (§8.5/14) is augmented for this case to remove any reference to conversion functions. This is further described in §15.3.3.
- If the initialization is taking place in base or member initializer and the destination type is a CLI class type, only constructors of the destination type are considered. The C++ Standard (§8.5/14) is augmented for this case to remove any reference to conversion functions.

14. Conversions

14.1 Conversion sequences

To accommodate the addition of boxing conversions and parameter array conversions, §13.3.3.2 of the C++ Standard is augmented, as follows:

> When comparing the basic forms of implicit conversion sequences (as defined in 13.3.3.1)
>
> - a standard conversion sequence (13.3.3.1.1) is a better conversion sequence than a boxing conversion sequence, a user-defined conversion sequence, a parameter array conversion sequence, or an ellipsis conversion sequence, and
> - a boxing conversion sequence is a better conversion sequence than a user-defined conversion sequence, a parameter array conversion sequence, or an ellipsis conversion sequence, and
> - a user-defined conversion sequence (13.3.3.1.2) is a better conversion sequence than a parameter array conversion sequence or an ellipsis conversion sequence (13.3.3.1.3).
> - a parameter array conversion sequence is a better conversion sequence than an ellipsis conversion sequence (13.3.3.1.3).

14.2 Standard conversions

The standard conversions in the C++ Standard apply to C++/CLI. C++/CLI has the following standard conversions as well.

14.2.1 Handle conversions

A handle conversion is similar to a pointer conversion as defined in the C++ Standard (§4.10). To accommodate the addition of handle conversions, Table 9, "conversions", in the C++ Standard, §13.3.3.1.1, "Standard conversion sequences", is augmented by the addition of a "Handle conversion" row, as shown in §18.3.

An rvalue of type “handle to *cv* `D`,” where `D` is a type, can be converted to an rvalue of type “handle to *cv* `B`,” where `B` is a base class of `D`. The result of the conversion is a handle to the same object.

Since the type `void^` is ill-formed, there is no handle conversion to it.

A handle to a type `array<S^,n>` has a handle conversion to a handle to type `array<T^,n>` provided `S^` has a handle conversion to `T^` and `n` (the rank of both CLI arrays) is the same. Such a conversion is better than a conversion from type `array<S^,n>` to `System::Array^`. This relationship is known as ***array covariance***. Because array covariance can allow a variable to refer to a base class of the array’s element type, assignments to elements of handle type arrays include a run-time check performed by the CLI (see CLI Partion III, §4.26 and §4.27). The run-time check ensures that the value being assigned to the array element is of a permitted type. Array covariance specifically does not extend to CLI arrays of value types. For example, no conversion permits an `array<int>` to be treated as `array<Object^>`.

A handle can be used as the first operand of a conditional operator.

The null value constant can be converted to any handle type; the result is a handle with ***null value*** of that type, and is distinguishable from every other value that is a handle to an CLI heap-based object. To support this, the C++ Standard is augmented, as follows:

> §4/2: [*Note*: … — When used in the condition of an `if` statement or iteration statement (6.4, 6.5). If the condition is a handle, and conversion from the handle to `bool` is not possible, the destination

type is the handle type; otherwise, the destination type is `bool`. If the condition is not a handle type, the destination type is `bool`. … *end note*]

§5.16/1: The first expression is implicitly converted to `bool` (clause 4). If that conversion is ill-formed and the expression is a handle type or a type given by a generic type parameter not constrained by the value class constraint, the expression is tested for the null value, returning true if not null and false if it is null. Otherwise, if the conversion to bool is ill-formed and the expression is not a handle type or a type given by a generic type parameter not constrained by the value class constraint, the program is ill-formed.

§6.4/4: The value of a *condition* that is an initialized declaration in a statement other than a `switch` statement is the value of the declared variable implicitly converted to type `bool`. ~~If that conversion is ill-formed, the program is ill-formed. The value of a condition that is an initialized declaration in a switch statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted to integral or enumeration type otherwise. The value of a condition that is an expression is the value of the expression, implicitly converted to bool for statements other than switch; if that conversion is ill-formed, the program is ill-formed. The value of the condition will be referred to as simply "the condition" where the usage is unambiguous.~~ The value of a condition that is an expression is the value of the expression, implicitly converted to bool for statements other than switch. If that conversion is ill-formed and the expression is a handle type or a type given by a generic type parameter not constrained by the value class constraint, the expression is tested for the null value, returning true if not null and false if it is null. Otherwise, if the conversion to bool is ill-formed and the expression is not a handle type or a type given by a generic type parameter not constrained by the value class constraint, the program is ill-formed. [*Note*: If there is no conversion to bool and the declared variable or expression is not a handle type, a conversion to a handle type is not considered. *end note*.]

§6.5.2/1: The expression is implicitly converted to bool; if that is not possible, and the expression is a handle type or a type given by a generic type parameter not constrained by the value class constraint, it is tested for null. If there is no conversion to bool, and the expression is not a handle type or a type given by a generic type parameter not constrained by the value class constraint, the program is ill-formed.

14.2.1.1 Ranking handle conversions

Of the additional standard conversion C++/CLI adds, only handle conversions can require further ranking to determine whether one conversion is better than another. In addition to the rules in the C++ Standard §13.3.3.2/4, the following rules apply:

- If class `B` is derived directly or indirectly from class `A` and class `C` is derived directly or indirectly from `B`,
  - Conversion of `C^` to `B^` is better than conversion of `C^` to `A^`.
  - Conversion of `B^` to `A^` is better than conversion of `C^` to `A^`.

14.2.2 Pointer conversions

The definition of ***null pointer constant*** in the C++ Standard (§4.10/1) is augmented, as follows:

> "A null pointer constant is either an integral constant expression rvalue of integer type that evaluates to zero, or the null value constant `nullptr`."

[*Note:* The implication of this is that the null value constant can be converted to any pointer type. *end note*]

The following conversion rules apply to interior pointers:

Conversion from `interior_ptr<T1>` to `interior_ptr<T2>` is allowed if and only if conversion from `T1*` to `T2*` is allowed;

In conversions between types where exactly one type is `interior_ptr<T1>`, the interior pointer behaves exactly as if it were "pointer to cv `T1`", with two exceptions:

- Conversion to any other type "pointer to cv `T1`" is not allowed. In particular, conversion from `interior_ptr<T>` to `T*` is not allowed.
- Conversion from the null pointer constant to `interior_ptr<T>` is not allowed, but conversion from the null value constant is allowed.

[*Example:*

```
array<int>^ arr = gcnew array<int>(100);
interior_ptr<int> ipi = &arr[0];
int* p = ipi;        // error; no conversion from interior to non-
interior
int k = 10;
ipi = &k;            // OK; k is an auto variable
ipi = 0;             // error; must use nullptr instead
ipi = nullptr;       // OK
ipi = p;             // OK
if (ipi) { … }       // OK
```

end example]

The following conversion rules apply to pinning pointers:

Conversion from `pin_ptr<T1>` to `pin_ptr<T2>` is allowed if and only if conversion from `T1*` to `T2*` is allowed;

In conversions between types where exactly one type is cv `pin_ptr<T>`, the pinning pointer behaves exactly as if it were "pointer to cv `T`", with the exception that conversion from a null pointer constant to `pin_ptr<T>` is not allowed, but conversion from the null value constant is allowed. [*Note:* In particular, conversion from `pin_ptr<T>` to `T*` is allowed as a standard conversion. *end note*]

[*Example:*

```
array<int>^ arr = gcnew array<int>(100);
pin_ptr<int> ppi = &arr[0];
int* p = ppi;           // OK
int k = 10;
ppi = &k;               // OK; k is an auto variable
ppi = 0;                // error; must use nullptr instead
ppi = nullptr;          // OK
pin_ptr<int> ppi2 = p;  // OK
```

end example]

14.2.3 Lvalue conversions

There is a standard conversion for each of the following: "cv-qualified lvalue of type T" to "cv-qualified gc-lvalue of type T," and "cv-qualified gc-lvalue of type T" to "cv-qualified rvalue of type T." If a cv-qualified lvalue would not convert to an rvalue in a given context, it is ill-formed for a gc-lvalue to convert to an rvalue. [*Rationale*: Conversion from a gc-lvalue to an rvalue when binding a native reference to an integer on the CLI heap results in loss of type safety. *end rationale*]

14.2.4 Integral promotions

To accommodate the addition of extended integer types, the C++ Standard (§4.5/1) is is augmented, as follows:

> An rvalue of type ~~`char`, `signed char`, `unsigned char`, `short int`, or `unsigned short int`~~ an integer type whose integer conversion rank (4.13) is less than the rank of `int` and `unsigned int` can be converted to an rvalue of type `int` if `int` can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type `unsigned int`.

and the C++ Standard is augmented by the following new clause, 4.13:

> 4.13 Integer conversion rank
>
> Every integer type has an *integer conversion rank* defined as follows:

• No two signed integer types shall have the same rank, even if they have the same representation.

• The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.

• The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.

• The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

• The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

• The rank of `char` shall equal the rank of `signed char` and `unsigned char`.

• The rank of `bool` shall be less than the rank of all other standard integer types.

• The rank of any enumerated type shall equal the rank of its underlying type (7.2).

• The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

• For all integer types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

[*Note*: The integer conversion rank is used in the definition of the integral promotions (4.5) and the usual arithmetic conversions (5).]

To accommodate the addition of the types `long long int` and `unsigned long long int`, the C++ Standard (§4.5/2) is augmented, as follows:

> "An rvalue of type `wchar_t` (3.9.1) or `System::Char` can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: `int`, `unsigned int`, `long`, ~~or~~ `unsigned long`, `long long int`, or `unsigned long long int`. An rvalue of an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of the enumeration (i.e., the values in the range b_{min} to b_{max} as described in 7.2): `int`, `unsigned int`, `long`, ~~or~~ `unsigned long`, `long long int`, or `unsigned long long int`."

14.2.5 String literal conversions

An rvalue of type <narrow-string-literal-type> can be converted to one of two types: `System::String^` or "array of *n* `const char`". When a <narrow-string-literal-type> is converted to `System::String^`, the result is treated as a CLI string literal (§34.4.1). When a <narrow-string-literal-type> is converted to an array, *n* is the size of the string (as defined in the C++ Standard, §2.13.4/5), the array has static storage duration, and the array is initialized with the given characters. A conversion from <narrow-string-literal-type> to `System::String^` is better than a conversion from <narrow-string-literal-type> to "array of *n* `const char`".

An rvalue of type <wide-string-literal-type> can be converted to one of two types: `System::String^` or "array of *n* `const wchar_t`". When a <wide-string-literal-type> is converted to `System::String^`, the result is treated as a CLI string literal (§34.4.1). When a <wide-string-literal-type> is converted to an array, *n* is the size of the string (as defined in the C++ Standard, §2.13.4/5), the array has static storage duration, and the array is initialized with the given characters. A conversion from <wide-string-literal-type> to `System::String^` is better than a conversion from <wide-string-literal-type> to "array of *n* `const wchar_t`".

For conversion in the presence of the subscript operator, see §15.3.1; for the unary `*` operator, see §15.4.1.2; for the binary `->` operator, see §15.3.4; and with the binary + operator, see §15.6.3.

Consider the case in which a function, whose *parameter-declaration-clause* terminates with an ellipsis, is called with a string literal as the argument that corresponds to the ellipsis. If the string literal is a narrow string literal, it is converted to an array of *n* `char`; if it is a wide string literal, it is converted to an array of *n* `wchar_t`.

14.2.6 Boxing conversions

A boxing conversion involves the creation of a new object on the CLI heap. A boxing conversion shall be applied only to instances of value types, with the exception of pointers. For any given value type `V`, the conversion results in a `V^`. [*Note*: Boxing in some other CLI-based languages goes directly from `V` to `Object^`. This can be achieved in C++/CLI via a boxing conversion followed by a handle conversion. *end note*] Although the value type expression can be cv-qualified, the resulting boxed value type is not.

To accommodate the addition of boxing conversions, Table 9, "conversions", in the C++ Standard, §13.3.3.1.1, "Standard conversion sequences", is augmented by the addition of a "Boxing conversion" row, as shown in §18.3. [*Example*: Note that the positioning of the boxing conversion in that table means that given a choice between a “narrowing” conversion and boxing, boxing is preferred. Given the following,

```
void F(float f) {
   Console::WriteLine("F(float)");
}
void F(Object^ o) {
   Console::WriteLine("F(Object^)");
}
int main() {
  F(3.14);
}
```

the output is "`F(Object^)`". *end example*]

A boxing conversion cannot be rewritten by the user; it is reserved to the implementation.

A boxing conversion follows the exact same sequence of operations as user-defined conversions (C++ Standard §13.3.3.1.2). Boxing conversions are considered before user-defined conversions, and a boxing conversion sequence never invokes a user-defined conversion. In other words, given a choice between applying a boxing conversion or a user-defined conversion, the boxing conversion is selected. Thus, §13.3.3.2 of the C++ Standard is augmented, as shown in §14.1 .

[*Note*: One can write a user-defined conversion operator that performs the same conversion as a boxing conversion. Although the compiler would not call this user-defined conversion in boxing contexts, the programmer could call the user -defined conversion using explicit operator function syntax. *end note*]

For metadata details, see §34.4.2.

14.3 Implicit conversions

14.3.1 Implicit constant expression conversions

The following implicit constant expression conversions are permitted:

- The null value constant can be converted to any pointer type.
- The null value constant can be converted to any handle type.

14.3.2 User-defined implicit conversions

14.3.3 Boolean Equivalence

Whether or not `bool` maps to `System::Boolean`, an rvalue of type `bool` can be converted to an rvalue of type `System::Boolean`, and an rvalue of type `System::Boolean` can be converted to an rvalue of type `bool`.

14.4 Explicit conversions

The following explicit conversions are permitted:

- The null value constant can be converted to any pointer type.
- The null value constant can be converted to any handle type.

14.5 User-defined conversions

Generic conversion functions are allowed. [*Note*: However, the need to check generic constraints after overload resolution makes it difficult to write a generic conversion that is useful. A template conversion function will usually be more useful. *end note*]

14.5.1 Constructors

Although the `explicit` keyword is permitted on a constructor in a ref class or value class, it has no effect. Constructors in these classes are never used for conversions or casts (see §13.3).

14.5.2 Explicit conversion functions

C++/CLI allows the `explicit` keyword on conversion functions. Thus, C++ Standard §7.1.2 is augmented, as follows:

> "The `explicit` specifier shall be used only in declarations of constructors within a class declaration, or on declarations of conversion functions within a class declaration; see 12.3.1 and 12.3.2."

A conversion function that is declared with the `explicit` keyword is known as an ***explicit conversion function***. A conversion function that is declared without the `explicit` keyword (i.e., every conversion function in Standard C++) is known as an ***implicit conversion function***.

Like an explicit constructor, an explicit conversion function can only be invoked by direct-initialization syntax (C++ Standard §8.5) and casts (C++ Standard §5.2.9, §5.4).

A type shall not contain an implicit conversion function and an explicit conversion function that perform the same conversion. Only one of these is allowed.

It is possible to write a class that has both an explicit converting constructor and a conversion function that can perform the same conversion. In this case, the explicit conversion function is preferred.

14.5.3 Static conversion functions

C++/CLI allows conversion functions, both implicit and explicit, to be `static`. Conversion functions shall not have namespace scope. A static conversion function shall take only one parameter, which is the type to convert from (a non-static member conversion function shall have no parameters). Neither static nor non-static conversion functions shall specify return types.

Either the source type (parameter type) or the target type (*type-specifier-seq*) is required to be `T`, `T^`, `T&`, `T%`, `T^%`, or `T^&`, where `T` is the type of the containing class. (`T*` is not allowed because conversions are not looked up through pointers.)

Implicit conversions can now be found in more than one place: the scope of the type of the source expression and the scope of all potential target types. If overload resolution results in a set of conversion functions (and possibly converting constructors) that can perform the same conversion, the program is ambiguous and ill-formed.

14.6 Parameter array conversions

The parameter array conversion sequence occurs when overload resolution chooses a function that takes a parameter array as its last argument. Such overloads are preferred to C-style variable-argument functions, and are not preferred to any other overloads.

A parameter array overload is chosen by overload resolution. For the purpose of overload resolution, the compiler creates signatures for the parameter array functions by replacing the parameter array argument with `n` arguments of the CLI array's element type, where `n` matches the number of arguments in the function call. These synthesized signatures have higher cost than other non-synthesized signatures, and they have lower cost than functions whose *parameter-declaration-clause* terminates with an ellipsis. [*Note*: This is similar to the tiebreaker rules for template functions and non-template functions in the C++ Standard (§13.3.3). *end note*]

For example, for the function call `f(var1, var2, …, varm, val1, val2, …, valn)`

```
void f(T1 arg1, T2 arg2, …, Tm argm, ... array<T>^ arr)
```

is replaced with

```
void f(T1 arg1, T2 arg2, …, Tm argm, T t1, T t2, …, T tn)
```

Overload resolution is performed with the set containing the synthesized signatures according to the rules of Standard C++. If overload resolution selects a C-style variable-argument conversion, it means that none of the synthesized signatures was chosen.

If overload resolution selects one of the synthesized signatures, the conversion sequences needed for each argument to satisfy the call is performed. For the synthesized parameter array arguments, the compiler constructs a CLI array of length *n* and initializes it with the converted values. Then the function call is made with the constructed parameter array.

[*Note*: User-defined conversions are better than parameter array conversions.

```
ref class A {};
ref class B {
public:
   static operator A^(B^ b) { return gcnew A; }
};
void F(... array<B^>^ arr) { Console::WriteLine("array<B^>^"); }

void F(A^ a) { Console::WriteLine("A^"); }
int main() {
  B^ b = gcnew B;
  F(b);
}
```

The program prints "A^". *end note*]

14.7 Naming conventions

During compilation, the name of the conversion function is the C++ identifier used in source code for that function. For example, the conversion function from `A` to `B` could be the static member function of either `A` or `B`, `operator B(A)`, or the instance function of `A`, `operator B()`. [*Example*:

```
public value struct Decimal {
   …
   static operator Decimal(int value);
   static explicit operator double(Decimal value);

   explicit operator float();
};
```

end example]

A program that declares or defines a member function within a ref class, value class, or interface class using the names `op_Implicit` or `op_Explicit`, is ill-formed. A program shall not directly refer to these names.

Operator functions are either CLS-compliant or C++-dependent.

A conversion function is *CLS-compliant* when all of the following conditions occur:

- The conversion function is a static member of a ref class or a value class.

- If a value class is a parameter or a target value of the conversion function, the value class shall not be passed by reference nor passed by pointer or handle.
- If a ref class is a parameter or a target value of the operator function, the ref class shall be passed by handle. The handle shall not be passed by reference.

If a conversion function does not match these criteria, it is *C++-dependent*.

15. Expressions

To accommodate the addition of the types `long long int` and `unsigned long long int`, and extended integer types, the C++ Standard (§5/9) is augmented as follows:

> Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions, which are defined as follows:
>
> — If either operand is of type `long double`, the other shall be converted to `long double`.
>
> — Otherwise, if either operand is `double`, the other shall be converted to `double`.
>
> — Otherwise, if either operand is `float`, the other shall be converted to `float`.
>
> ~~— Otherwise, the integral promotions (4.5) shall be performed on both operands.~~
>
> ~~— Then, if either operand is `unsigned long` the other shall be converted to `unsigned long`.~~
>
> ~~— Otherwise, if one operand is a `long int` and the other `unsigned int`, then if a `long int` can represent all the values of an `unsigned int`, the `unsigned int` shall be converted to a `long int`; otherwise both operands shall be converted to `unsigned long int`.~~
>
> ~~— Otherwise, if either operand is `long`, the other shall be converted to `long`.~~
>
> ~~— Otherwise, if either operand is `unsigned`, the other shall be converted to `unsigned`.~~
>
> ~~[*Note*: otherwise, the only remaining case is that both operands are `int`]~~
>
> — Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
>
> — If both operands have the same type, then no further conversion is needed.
>
> — Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
>
> — Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
>
> — Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
>
> — Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

15.1 Function members

The following function member kinds are added to those defined by Standard C++:

- Properties (both scalar and default-indexed)
- Events

The statements contained in these function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

Invocations of default-indexed properties employ overload resolution to determine which of a candidate set of function members to invoke.

[*Note:* The following table summarizes the processing that takes place in constructs involving these three categories of function members that can be explicitly invoked. In the table, `e`, `x`, `y`, and `value` indicate expressions classified as variables or values, `E` is an event, and `P` is the simple name of a property.

Construct	Example	Description
Property access	`P`	`P::get()`
	`P = value`	`P::set(value)`
Event access	`E += value`	`E::add(value)`
	`E -= value`	`E::remove(value)`
Default-indexed property access	`e[x, y]`	`e.default::get(x, y)`
	`e[x, y] = value`	`e.default::set(x, y, value)`

end note]

15.2 Primary expressions

To accommodate the addition of properties, the "Primary expressions" subclause of the C++ Standard (§5.1) is augmented, as follows:

> "A static property or event is not associated with any instance of a class, and a program is ill-formed if it refers to `this` in the accessor functions of a static property or event."
>
> "An instance property or event is associated with a specific instance of a class, and that instance can refer to `this` in the accessor functions of that instance property or event."

15.3 Postfix expressions

To accommodate the addition of default-indexed properties and CLI arrays (which are accessed using subscript-like expressions), the C++ Standard grammar (§5.2) for *postfix-expression* is augmented, as follows:

postfix-expression:
primary-expression
postfix-expression `[` *expression-list* `]`
postfix-expression `(` *expression-list*$_{opt}$ `)`
simple-type-specifier `(` *expression-list*$_{opt}$ `)`
`typename` `::`$_{opt}$ *nested-name-specifier* *identifier* `(` *expression-list*$_{opt}$ `)`
`typename` `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id* `(` *expression-list*$_{opt}$ `)`
postfix-expression `.` *template*$_{opt}$ *id-expression*
postfix-expression `->` *template*$_{opt}$ *id-expression*
postfix-expression `.` *pseudo-destructor-name*
postfix-expression `->` *pseudo-destructor-name*
postfix-expression `++`
postfix-expression `--`
dynamic_cast `<` *type-id* `>` `(` *expression* `)`
`static_cast` `<` *type-id* `>` `(` *expression* `)`
`reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
`const_cast` `<` *type-id* `>` `(` *expression* `)`
`typeid` `(` *expression* `)`
`typeid` `(` *type-id* `)`
`typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier* *identifier* `::` `typeid`
`typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id* `::` `typeid`

The C++ Standard production

postfix-expression `[` *expression* `]`

is augmented to

postfix-expression `[` *expression-list* `]`

to accommodate indexed access (§15.3.1) and CLI array element access (§24.3). As a result, commas in square-bracketed expressions are not operators and instead are list separators.

To allow constructs such as `List<List<int>>`, where >> is treated as two tokens instead of one, the C++ Standard (§5.2/2) is augmented by the following new paragraph:

> [*Note*: The > token following the *type-id* in a `const_cast`, `dynamic_cast`, `reinterpret_cast`, `safe_cast`, or `static_cast` may be the product of replacing a >> token by two consecutive > tokens (14.2). *end note*]

15.3.1 Subscripting and indexed access

The subscripting operator `[]` can represent the built-in subscripting operator (C++ Standard §5.2.1), a call of an overloaded `operator[]` (C++ Standard §13.5.5), or a use of an indexed property. Overload resolution is used to determine which applies. As in the C++ Standard, if neither operand is a class or enum or a handle to a class, overload resolution is not needed and the built-in operator is selected.

For any given instance of a ref class, subscripting can be applied to that instance and to a handle to that instance, with the same result.

The argument list for the overload resolution is the left operand plus the list of expressions of the expression-list. [*Note*: in Standard C++, the syntactic term inside the `[]` is an expression, which means that `x[i,j]` is a valid subscripting operation whose subscript is a comma-expression (in other words, it's effectively `x[j]`). In C++/CLI, a top-level comma inside `[]` is considered a list separator and not an operator, so `x[i,j]` would only match an indexed property taking two arguments. If one wants a top-level comma operator, one must write it inside parentheses, e.g., `x[(i,j)]`. This is true even when `x` does not have class type or handle to class type. *end note*]

A CLI class type shall not have both a default-indexed property and an `operator[]`.When subscript is applied to a string literal, that literal is converted to an "array of *n* `const char`" or "array of *n* `const wchar_t`", as appropriate. The following built-in operator functions exist:

```
const char& operator[](<narrow-string-literal-type>, integer-type);
const wchar_t& operator[](<wide-string-literal-type>, integer-type);
const char& operator[](integer-type, <narrow-string-literal-type>);
const wchar_t& operator[](integer-type, <wide-string-literal-type>);
```

where *integer-type* is any integer type.

15.3.2 Function call

The C++ Standard (§5.2.2/1) states, “A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of expressions, which constitute the arguments to the function.”

C++/CLI contains support for delegates (§27). As such, the postfix expression can be a delegate type, in which case, the whole expression is a delegate invocation (§27.3), and the argument list is passed to each function encapsulated by the delegate.

15.3.3 Explicit type conversion (functional notation)

Function-style casts of ref classes and value classes do not invoke conversions; these are calls to constructors only. If a corresponding constructor does not exist, the program is ill-formed. [*Example*:

```
value class C {};

value class E {
public:
  operator C() { return C(); }
};

void F(C c) {}

int main() {
  E e;
  F(C(e));  // error - no constructor of C matches parameter
}
```

end example]

15.3.4 Class member access

To accommodate the use of handles with `->`, the text in Standard C++ (§5.2.5/2) is augmented, as follows:

> "For the second option (arrow) the type of the first expression (the pointer expression) shall be "handle to class object" (of a complete type) or "pointer to class object" (of a complete type)."

The text in Standard C++ (§5.2.5/3) is amended, as follows:

> "If `E1` has the type "pointer to class `X`," then the expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`. If `E1` has the type "handle to class `X`", and `X` has an `operator->` the expression `E1->E2` is evaluated as `(*(E1)).operator->(E2)`. Otherwise, if `E1` has the type "handle to class `X`" and `X` does not have an `operator->`, then the expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`."

and footnote 59 is augmented, as follows:

> "59) Note that if `E1` has the type "pointer to class `X`", then `(*(E1))` is an lvalue. If `E1` has the type "handle to class `X`", then `(*(E1))` is a gc-lvalue."

If a program accesses an instance of a value type directly using the arrow operator, it is ill-formed. [*Note*: Applying the arrow operator to an instance of a value type does *not* box that value. However, certain accesses to such an instance using the dot operator require boxing. See the metadata details in §34.5.1. *end note*]

When a string literal is the left-hand operand to the binary `operator->`, that literal is converted to `System::String^`.

15.3.5 Increment and decrement

See §19.7.3.

15.3.6 Dynamic cast

For the expression `dynamic_cast<T>(e)`, in addition to the rules specified by the C++ Standard (§5.2.7), the following also applies:

If `T` is a tracking reference type, `e` shall be a gc-lvalue of a complete class type, and the result is a gc-lvalue of the type referred to by `T`.

`T` can be a handle type, and in such cases `e` shall be an rvalue of a handle to complete class type, and the result is an rvalue of type `T`.

If the value of `e` is a null value and `T` is handle type, the result is the null value of type `T`.

If T is "handle to cv1 `B`" and `e` has type "handle to cv2 `D`" such that `B` is a base class of `D`, the result is a handle to `B` such that it refers to the same CLI heap-based object as `e`. The *cv-qualification* for cv1 shall be the same as or greater than that for cv2. Otherwise, a runtime check is required. If the runtime check cannot succeed, the program is ill-formed.

If `T` is either a handle or a pointer to any type other than a native class, and the cast fails, the result is the null value or the required result type. If `T` is a reference to any type other than a native class and the cast fails, then the expression throws `System::InvalidCastException`. When `T` is a native class, the rules of Standard C++ §5.2.7/9 apply.

For metadata details, see §34.5.2.

15.3.7 Type identification

C++/CLI adds a new use of the `typeid` keyword, whereby a given type name can be followed by `::typeid` to get a `System::Type^` for the given type name. This construct is referred to here as a ***typeid Type expression*** (which is unrelated to Standard C++'s typeid expression). To accommodate this, the C++ Standard grammar production for *postfix-expression* (§5.2 and §A.4) is augmented (§15.3).

In the C++ Standard (§14.6.2.2/4), the "Expressions of the following forms" list is augmented to include the new typeid Type expression forms of *postfix-expression* (§15.3).

The result of a typeid Type expression is an lvalue of static type `System::Type^`. There is only one `System::Type` object for any given type. [*Note*: This means that for any type `T`, `T::typeid == T::typeid` is always true. *end note*] As this form is a compile-time expression, it can be used as an argument to an attribute constructor.

The type name in the typeid Type expression shall be a raw type (§12.3.1) or a pointer to a raw type.

The type in a typeid Type expression can be any handle `R^` provided that type is referred to via a typedef. The result of such an expression is the same as applying typeid directly to type `R`. The type `R%` is handled the same way.

Each fundamental type is a distinct type; however, different fundamental types can map to the same CLI type. As such, the `typeid` operator shall produce the same `Type` handle for each fundamental type that maps to the same CLI type, regardless of whether optional or required modifiers (§33.1) are otherwise required to distinguish those fundamental types. [*Example*: In an implementation in which `int` and `long` both map to `System::Int32`, both `int::typeid` and `long::typeid` result in a `Type^` describing `System::Int32`. *end example*]

[*Note*: The practice of using a lock on `T::typeid` to guard static members of a type `T` is discouraged, as it can lead to deadlock. *end note*]

The typeid Type expression provides convenient syntactic access to the functionality of the `System::Type::GetType()` library function. Whereas `GetType()` shall be called on an CLI heap-based object of the given type, `::typeid` can be applied to a type directly, and consequently does not require a CLI heap-based object to be created. [*Example*:

```
using namespace System::Reflection;

ref class X { … };

Console::WriteLine(X::typeid);   // does not require an object
X^ pX = gcnew X;
Type^ pType = pX->GetType();     // GetType requires an object
Console::WriteLine(pType);

Console::WriteLine(Int32::typeid);
Console::WriteLine(array<Int32>::typeid);
Console::WriteLine(void::typeid);

Type^ t = String::typeid;
Console::WriteLine(t->BaseType);

array<MethodInfo^>^ functions = t->GetMethods();
for each (MethodInfo^ mi in functions)
   Console::WriteLine(mi);
```

The output produced is:

```
X
X
System.Int32
System.Int32[]
System.Void
System.Object
…
System.CharEnumerator GetEnumerator()
System.Type GetType()
```

end example]

The `::typeid` operator can be applied to a type parameter or to a constructed type: the result is a CLI heap-based object of type `System::Type` that represents the runtime type of the type parameter or constructed type. Outside of the body of a generic type definition, the `::typeid` operator shall not be applied to the bare name of that type. [*Example*:

```
generic<typename T>
ref class X {
public:
   static void F() {
      Type^ t1 = T::typeid;      // okay
      Type^ t2 = X<T>::typeid;   // okay
      Type^ t3 = X::typeid;      // okay
   }
};

int main() {
   Type^ t4 = int::typeid;       // okay
   Type^ t5 = X<int>::typeid;    // okay
   Type^ t6 = X::typeid;         // error
}
```

Clearly, the initialization of `t6` is in error. However, that of `t3` is not, as the use of `X` is really an implicit use of `X<T>` (§31.1.2). *end example*]

The `::typeid` operator can be used in an argument to an attribute constructor call. [*Example*:

```
[AttributeUsage(AttributeTargets::All)]
public ref struct XAttribute : Attribute {
   XAttribute(Type^ t) {}
};

[X(int::typeid)]
public ref class R {};
```

end example]

Standard C++'s native typeid can be applied to *expression* or *type-id*. Native typeid shall not be used with types that are ref classes, interface classes, handles, value classes other than fundamental types, enums of any kind, or pointers. Thus, any program that contains a native typeid with *expression* or *type-id* having any of these types, is ill-formed.

15.3.8 Static cast

The rules specified by the C++ Standard (§5.2.9) apply. For the expression, `static_cast<T>(e)`, the following also applies.

A static cast can invoke a user-defined conversion function as described in the C++ Standard (§5.2.9/2). All of the following are considered: explicit conversion functions, implicit conversion functions, explicit converting constructors, and implicit converting constructors.

[*Note*: Non-native types do not have converting constructors. *end note*]

The cast expression discussed in the C++ Standard (§5.2.9/3) is also allowed on tracking references.

The conversion discussed in the C++ Standard (§5.2.9/7) is allowed for both native and CLI enumerations.

An rvalue of type "handle to cv1 `B`", where `B` is a type, can be converted to an rvalue of type "handle to cv2 `D`", where `D` is a class derived from `B`, if a valid standard conversion from "handle to `D`" to "handle to `B`" exists (§14.2.1), and cv2 is the same *cv-qualification* as, or greater *cv-qualification* than, cv1. The null value is converted to the null value of the destination type.

15.3.9 Reinterpret cast

The rules of specified by the C++ Standard (§5.2.10) apply. A reinterpret cast expression that attempts to cast from or to a handle type is ill-formed.

A reinterpret cast will never invoke a boxing conversion sequence.

15.3.10 Const cast

The rules specified by the C++ Standard (§5.2.11) apply. For the expression, `const_cast<T>(v)`, the following also applies.

Where the C++ Standard discusses the application of `const_cast` to pointers, the rules shall also apply to handles.

An lvalue of type `T1` can be explicitly converted to an lvalue of type `T2` using the cast `const_cast<T2%>` if a pointer or handle to `T1` can be explicitly converted to the type pointer or handle to `T2` using a `const_cast`. The result of a reference `const_cast` refers to the original object.

A null value is converted to the null value of the destination type. A program in which `v` in the const cast expression is the `nullptr` literal is ill-formed.

A const cast shall never invoke a boxing conversion sequence.

15.3.11 Safe cast

A safe cast performs the optimal cast for frameworks programming. The compiler processes a `safe_cast` expression as follows:

- The compiler performs a lookup in the current context for the name `safe_cast`.
- If the name refers unambiguously to `::cli::safe_cast`, or the name is not found, then the expression is processed by the compiler according to the following grammar, and interpreted according to the rules specified herein.

`safe_cast` `<` *type-id* `>` `(` *expression* `)`

The result of the expression `safe_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is a tracking reference type, the result is a gc-lvalue; otherwise, the result is an rvalue. Types shall not be defined in a `safe_cast`. The `safe_cast` operator shall not cast away constness. The type `T` and the type of `v` shall not be a native class, a pointer, a pointer-to-member, a native reference, or an indirection to a native class, pointer, or pointer-to-member. [*Note*: Except for the cases just mentioned, a `safe_cast` in which the target type or the type of the expression is anything else is always verifiable. An explicit type conversion—also known as a C-style cast—always defaults to safe cast behavior when the arguments allow the generation of verifiable code for the conversion. *end note*]

An expression `e` can be explicitly converted to a type `T` using a `safe_cast` of the form `safe_cast<T>(e)` if the declaration "`T t(e);`" is well-formed, for some invented temporary variable `t`. The effect of such an explicit conversion is the same as performing the declaration and initialization and then using the temporary variable as the result of the conversion. The result is a gc-lvalue if `T` is a tracking reference type, and an rvalue otherwise. The expression `e` is used as a gc-lvalue if and only if the initialization uses it as a gc-lvalue.

Otherwise, the `safe_cast` shall perform one of the conversions listed below. No other conversion shall be performed explicitly using `safe_cast`.

The inverse of any standard conversion sequence, other than the lvalue-to-rvalue, array-to-pointer, function-to-pointer, pointer conversions, pointer-to-member conversions, and Boolean conversion, can be performed

explicitly using `safe_cast`. Such a `safe_cast` is subject to the restriction that the explicit conversion does not cast away constness, and the following addition rules for specific cases:

- A value of integral or enumeration type can be explicitly converted to an enumeration type. The value is unchanged if the original value is within the range of the enumeration values. Otherwise, the resulting enumeration value is unspecified.
- If `T` is "handle to cv1 `D`", and the type of `v` is "handle to cv2 `B`", cv1 shall have the same cv-qualification as, or greater cv-qualification than, cv2, and a run-time check is applied to determine that `D` inherits from `B`. (For metadata and result details, see §34.5.1.) A `System::InvalidCastException` is thrown if the conversion fails. In the handle case, if the value of `v` is a null value, the result is the null value of type `T`. If the conversion cannot succeed at runtime, the program is ill-formed. [*Example*: if two ref classes `A` and `B` are unrelated, and the program uses `safe_cast<A^>(b)` where `b` has type `B^`, the dynamic check cannot succeed. *end example*]
- If `T` is "tracking reference to cv1 `D`", and the type of `v` is "cv2 `B`", cv1 shall have the same cv-qualification as, or greater cv-qualification than, cv2, and a run-time check is applied to determine that `D` inherits from `B`. (For metadata and result details, see §34.5.1.) A `System::InvalidCastException` is thrown if the conversion fails. If the conversion cannot succeed at runtime, the program is ill-formed.
- An rvalue of type "handle to cv1 `R`" can be converted to an lvalue of type `V`, where `V` is a value type. `R` shall be `System::Object`, `System::ValueType`, or an interface that `V` implements. If `V` is an enumeration type, `R` can also be `System::Enum`. (For metadata and result details, see §34.5.1.) A `System::InvalidCastException` is thrown if the conversion fails. This conversion sequence is called ***unboxing***. [*Note*: `safe_cast` is the only cast that can result in unboxing. *end note*]

15.4 Unary expressions

15.4.1 Unary operators

15.4.1.1 Unary &

When applied to an lvalue of type `T`, & yields a `T*` (see Standard C++ §5.3.1/2). When applied to a gc-lvalue of type T, & yields an `interior_ptr<T>` (§12.3.6).

A program that attempts to apply the built-in unary & operator to an instance of a ref class type, a literal field, or to a property, or to an initonly field outside of the class's constructor, is ill-formed.

A program that attempts to take the address of a member function of a non-native class in any context other than in the creation of a delegate, is ill-formed. There is no pointer-to-member representation for members of non-native classes. [*Example*:

```
delegate void D(int i);

ref struct R {
   static  void M1(int a) { }
           void M2(int b) { }
   virtual void M3(int c) { }
};

int main() {
   R^ r = gcnew R;
   D^ d;
   d =  gcnew D(&R::M1);
   d =  gcnew D(r, &R::M2);
   d += gcnew D(r, &R::M3);
}
```

end example]

For details on the metadata for delegate creation, see §34.14.

15.4.1.2 Unary *

The C++ Standard (§5.3.1/1) is augmented to allow for indirection on handles. Specifically, the following text:

> The unary * operator performs ***indirection***: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is "pointer to `T`," the type of the result is "`T`."

has been replaced with:

> The unary * operator performs ***indirection***: the expression to which it is applied shall be one of the following:
>
> - If the expression is a pointer to an object type or a pointer to a function type, then the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is "pointer to `T`," the type of the result is "`T`."
> - If the expression is a handle to an object, then the result is a gc-lvalue referring to the object to which the expression points. If the type of the expression is "handle to T," the type of the result is "`T`."

Dereferencing a `T^` yields a gc-lvalue of type `T`.

When `operator*` is applied to a string literal, that literal is converted to an "array of *n* `const char`" or "array of *n* `const wchar_t`", as appropriate. The following built-in operator functions exist:

```
const char& operator*(<narrow-string-literal-type>);
const wchar_t& operator*(<wide-string-literal-type>);
```

[*Note*: Because user-defined operators can work on handles, when a ref or value class has a user defined instance unary operator *, dereferencing a handle to such a class will invoke the user defined operator rather than actually dereferencing the handle. This is because all instance operators work on the class type as well as on a handle to the class (Standard C++ §19.7.1). For example:

```
ref struct R {
   int operator*() {
      Console::WriteLine("R::operator*");
      return 42;
   }
};
int main() {
   R^ r1a = gcnew R;
   int x = *r1a;  // calls operator*()
   R r1b;
   x = *r1b;      // calls operator*()
}
```

As this may be surprising to programmers, a quality implementation should warn when a ref class or value class has an instance operator *. The preferred alternative to such an operator is a pair of static operators, so that the operand is clearly stated to be either the class type or a handle to the class type, as follows:

```
ref struct R {
   static int operator*(R^ r) {
      Console::WriteLine("R::operator*(R^)");
      return 42;
   }
   static int operator*(R% r) {
      Console::WriteLine("R::operator*(R%)");
      return 42;
   }
};
```

```
int main() {
   R^ r2a = gcnew R;
   int x = *r2a;  // calls operator*(R^)

   R r2b;
   x = *r2b;   // calls operator*(R%)
}
```

end note]

15.4.1.3 Unary %

The result of the unary % operator is a handle to its operand, which, ordinarily, shall be a gc-lvalue. However, if the operand is an instance of a value class, the operand can be an rvalue. If the type of the expression is "`T`", and `T` is not a value class, the result is an rvalue and its type is "handle to `T`." In particular, the result of getting a handle of an object of type "*cv* `T`" is "handle to *cv* `T`," with the same cv-qualifiers. If `T` is a value class, the expression invokes the boxing conversion sequence (which allows loss of cv-qualification), which results in an rvalue. [*Example:*

```
ref class R {};
value class V {};
void f(System::Object^ o) {}

void g() {
   R r;
   f(%r);
   V v;
   f(%v);      // v is boxed
}
```

end example]

[*Note:* All handles to the same CLI heap-based object compare equal. For value classes, because % is a boxing operation, multiple applications of % results in handles that do *not* compare equal. *end note*]

A program that applies the unary % operator to a native class type is ill-formed.

15.4.1.4 Unary ^

No such operator exists. [*Rationale*: As a result, there is asymmetry between %/^ and &/*, in that unary * is used to dereference both * and ^. However, allowing a single syntax to be used in the latter case permits the writing of agnostic templates and generics. In any event, adding this operator would provide no new semantics, and would preclude the addition of such an operator later on, with new semantics. *end rationale*]

15.4.1.5 Logical negation

The C++ Standard (§5.3.1/8) is augmented as follows:

The operand of the logical negation operator `!` is implicitly converted to `bool` (clause 4); its value is `true` if the converted operand is `false` and `false` otherwise. If the implicit conversion to `bool` is ill-formed and the operand is a handle type or a type given by a generic type parameter not constrained by the value class constraint, the value is `true` if the handle is null and `false` if the handle is not null. The type of the result is `bool`. [*Example:*

```
ref class R { … };
R^ r = …;

if (!r)
   // handle is null
else
   // handle is non-null
```

end example]

15.4.2 Increment and decrement

See §19.7.3.

15.4.3 Sizeof

The C++ Standard (§5.3.3/1) is augmented, as follows:

> The `sizeof` operator shall not be applied to an expression that has function or incomplete type, or to an enumeration type before all its enumerators have been declared, or to the parenthesized name of such types, or to an lvalue that designates a bit-field, or to an expression that has null type, or to a handle, or to a tracking reference, or to a ref class. `sizeof(char)`, `sizeof(signed char)` and `sizeof(unsigned char)` are 1; ~~the result of `sizeof` applied to any other fundamental type (3.9.1) is implementation-defined~~. [*Note*: in particular, `sizeof(bool)`, ~~and~~ `sizeof(wchar_t)`, `sizeof(short int)`, `sizeof(int)`, `sizeof(long int)`, `sizeof(long long int)`, `sizeof(float)`, `sizeof(double)`, and `sizeof(long double)` are implementation-defined. *end note*]

C++ Standard (§5.3.3/2) is augmented by the addition of the following:

> When applied to a value class type, handle type, or generic type parameter, the result is not a compile-time constant expression. [*Note*: The definition of value class types excludes fundamental types and pointers, thus sizeof expressions on fundamental types and pointers are still compile-time constant expressions. *end note*]
>
> When applied to a ref class type or interface type, the program is ill-formed.

Due to requirements imposed by the CLI Standard, `size_t` shall be at least a 4-byte, unsigned integer.

15.4.4 New

A program is ill-formed if it attempts to allocate memory using `new` for an object of CLI class type other than a simple value class (§22.4).

15.4.5 Delete

The C++ Standard (§5.3.5/1) is augmented to allow for deletion of objects allocated on the CLI heap, as follows:

> The operand shall have a pointer type, a handle type, or a class type having a single conversion function (12.3.2) to a pointer type.
>
> In the first alternative (delete object), the value of the operand of delete shall be a pointer or handle to a non-array object or a pointer to a sub-object (1.8) representing a base class of such an object (clause 10).
>
> If the delete-expression calls the implementation deallocation function (3.7.3.2), and if the operand of the delete expression is not the null pointer constant, the deallocation function will deallocate the storage referenced by the pointer or handle thus rendering the pointer or handle invalid.

The array form of `delete` shall not be used on a handle type.

Inside of a generic, if an object's type is a generic type parameter, `delete` can be used to invoke that object's destructor. If the generic parameter type is constrained to the `System::IDisposable` interface, the delete expression evaluates to a call through that interface on the object. If the generic parameter type is not constrained to the `System::IDisposable` interface, the object is converted to `System::IDisposable^` using dynamic cast and the call is made through the converted object if the handle is not null. [*Note*: In the latter case, the conversion may require boxing if the generic type parameter can be a value type. Other than the negligible performance overhead of boxing and the ensuing dynamic cast to `IDisposable^`, calling the destructor on the boxed object will have no semantic impact on the program, as destructors on value types don't do anything (they cannot be defined by users). *end note*]

15.4.6 The gcnew operator

The `gcnew` operator is similar to the `new` operator, except that the former creates an object on the CLI heap. The type of the result of the `gcnew` operator is a handle to the type of the object allocated. In out-of-memory situations, `gcnew` throws `System::OutOfMemoryException`.

There is no array form of `gcnew`. There is no placement form of `gcnew`. The `gcnew` operator cannot be overloaded or replaced. There is no class-specific form of `gcnew`.

A program is ill-formed if it attempts to allocate memory for an object of native class type using `gcnew`.

In the C++ Standard (§5.3.4), a *new-expression* is used to allocate memory for an object at runtime. This grammar is augmented to accommodate the addition of the `gcnew` operator, as follows:

new-expression:
`::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id* *new-initializer*$_{opt}$
`::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$
`gcnew` *type-specifier-seq* *new-initializer*$_{opt}$ *array-init*$_{opt}$

In the `gcnew` case, the type of the object being allocated shall not be an abstract class type, nor shall it be incomplete. *array-init* shall only be used when creating a CLI array (see §24.2). [*Note:* The `gcnew` operator applied to a value class creates a boxed value. *end note*]

The `gcnew` operator is used to create an instance of a delegate. For more information, see §27.2.

15.4.7 The throw expression

As control passes from a *throw-expression* to a handler, *finally-clause*s, if any, are invoked for all *try-block* or *function-try-block*s entered since the *try-block* or *function-try-block* containing the handler was entered. The *finally-clause*s are invoked in the reverse order of the invocation of their parent *try-block* or *function-try-block*s.

The automatic destruction of objects in any given *try-block* or *function-try-block* required by the C++ Standard (15.2) takes place prior to the invocation of any *finally-clause* associated with that *try-block* or *function-try-block.*

For an example, see §16.4

If an object is thrown by handle (regardless of the kind of class to which the handle refers), the exception handling mechanism used shall be that defined by the CLI. (This includes boxed value types.) Otherwise, the Standard C++ mechanism shall be used.

Almost all types of objects can be thrown; exceptions to this rule are ref classes and value classes being thrown by value or by reference. It is always permitted to throw an object by handle. Other than stated in this Standard, the set of types that shall not be thrown using the CLI mechanism is the same as that for Standard C++.

A program that attempts to `throw nullptr` is ill-formed.

15.5 Explicit type conversion (cast notation)

The rules in the C++ Standard (§5.4/5) is augmented for C++/CLI by including safe casts before static casts.

- a `const_cast`
- a `safe_cast`
- a `safe_cast` followed by a `const_cast`
- a `static_cast`
- a `static_cast` followed by a `const_cast`
- a `reinterpret_cast`

- a `reinterpret_cast` followed by a `const_cast`

[*Note:* Standard C++ programs remain unchanged by this, as safe casts are ill-formed when either the expression type or target type is a native class. *end note*]

If both the type of the argument and the type being converted to are not a native class, a pointer, a pointer-to-member, a native reference, or an indirection to a native class, pointer, or pointer-to-member, then an explicit type conversion shall not use `static_cast` or `reinterpret_cast`. [*Note*: When arguments involve CLI class types, explicit type conversions always produce verifiable results. This enables programmers to use explicit type conversion syntax as the most suitable alternative for another language's cast notation. *end note*]

15.6 Additive operators

15.6.1 Delegate combination

Every delegate type provides the following predefined operator, where `D` is the delegate type:

```
static D^ operator +(D^ x, D^ y);
```

The binary `+` operator performs delegate combination when both operands are of the same delegate type D. The result of the operator is the result of calling `System::Delegate::Combine(x,y)`, and casting the result to `D^`. [*Note:* For examples of delegate combination, see §15.6.1 and §27.1. Since `System::Delegate` is not itself a delegate type, `operator+` is not defined for it. The behavior when either operand is `nullptr` is described in §27.1. *end note*]

15.6.2 Delegate removal

Every delegate type provides the following predefined operator, where `D` is the delegate type:

```
static D^ operator –(D^ x, D^ y);
```

The binary `-` operator performs delegate removal when both operands are of the same delegate type `D`. The result of the operator is the result of calling `System::Delegate::Remove(x,y)`, and casting the result to `D^`.

[*Note:* the `+=` and `-=` operator are defined via assignment operator synthesis (§19.7.4). The behavior when operand `y` is `nullptr` is described in §27.1. *end note*]

[*Example:*

```
delegate void D(int x);
ref struct Test {
   static void M1(int i) { … }
   static void M2(int i) { … }
};
int main() {
   D^ cd1 = gcnew D(&Test::M1);
   D^ cd2 = gcnew D(&Test::M2);
   D^ cd3 = cd1 + cd2;
   cd3 -= cd1;
   cd3 += cd1;
   cd3 = cd3 – (cd1 + cd2);
}
```

end example]

15.6.3 String concatenation

When the binary `operator+` is applied to a string literal, that literal is converted to `System::String^`. As a result, when a value having any integral type is added to a string literal, string concatenation results. [*Note*: This change in behavior from Standard C++ is intentional. *end note*]

The following built-in operator functions exist:

```
System::String^ operator+(<narrow-string-literal-type>, integer-type);
System::String^ operator+(<wide-string-literal-type>, integer-type);
System::String^ operator+(integer-type, <narrow-string-literal-type>);
System::String^ operator+(integer-type, <wide-string-literal-type>);
```

where *integer-type* is any integer type. When one of the operands to the binary + operator is a `System::String^`, string concatenation results. If the other operand does not also have type `System::String^`, its value is converted to that type by calling its `ToString` function. The following built-in operator functions exist:

```
System::String^ operator+(System::String^, System::String^);
System::String^ operator+(System::String^, System::Object^);
System::String^ operator+(System::Object^, System::String^);
```

[*Example*:

```
Point^ p = gcnew Point(5,6);
String^ s = "C++" + L"/CLI";  // s => "C++/CLI"
s = 3 + " apples";            // s => "3 apples"
s = "p is " + p;              // s => "p is (5,6)"
```

end example]

These three built-in functions can be hidden by user-defined versions. [*Example*: The program

```
String^ operator+(String^ l, String^ r) { return l; }

int main() {
   Console::WriteLine("ABC" + "DEF");
}
```

prints "`ABC`". *end example*]

A program containing an expression of the form *strlit* `-` *intexp*, where *strlit* is a string literal and *intexp* is any integer expression, is ill-formed.

15.7 Shift operators

To accommodate the addition of the types `long long int` and `unsigned long long int`, the C++ Standard (§5.8/2) is augmented, as follows:

> The value of `E1 << E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bit positions; vacated bits are zero-filled. If `E1` has an unsigned type, the value of the result is `E1` multiplied by the quantity 2 raised to the power `E2`, reduced modulo `ULLONG_MAX+1` if `E1` has type `unsigned long long int`, `ULONG_MAX+1` if `E1` has type `unsigned long`, `UINT_MAX+1` otherwise. [*Note*: the constants `ULLONG_MAX`, `ULONG_MAX`, and `UINT_MAX` are defined in the header `<climits>`). *end note*]

15.8 Relational operators

15.8.1 Handle equality operators

Every ref class type and value class type `C` implicitly provides the following predefined equality operators:

```
bool operator ==(C^ x, C^ y);
bool operator !=(C^ x, C^ y);
```

The implicity provided handle equality operators are used only if overload resolution finds no applicable equality operators (user-defined or otherwise defined in this specification). [*Example*: Delegates and `System::String` have equality operators defined already. If overload resolution selects one of those operators, the implicitly defined handle equality operators are not applicable. *end example*]

There are special rules for determining when a handle equality operator is applicable. For an *equality-expression* with operands of type `A^` and `B^`, define A_0 as follows:

- If `A` is a generic type parameter known to be a ref class, let A_0 be the effective base class of `A`.

- Otherwise, if A is an interface type, a ref class type, a value type other than pointers, or the null type, let A_0 be the same as A.
- Otherwise, no implicit handle equality operator is applicable.

Now define A_1 as follows:

- If A_0 is an interface type, a delegate type, `System::Delegate`, or `System::String`, let A_1 be `System::Object`.
- Otherwise, if A_0 is a CLI array type, let A_1 be `System::Array`.
- Otherwise, A_0 is the null type, a ref class type, or a value type other than pointer, and let A_1 be the same as A_0.

Define B_0 and B_1 in the same manner. Now determine if any implicit handle equality operators are applicable as follows:

- If both of the types A and B are the null type, then overload resolution is not performed and the result is constant `true` for `operator==` and `false` for `operator!=`.
- Otherwise, if there is no identity or handle conversion from A_0^ to B_0^ or no identity or handle conversion from B_0 to A_0, then no implicit handle equality operator is applicable.
- Otherwise, if there is an identity or handle conversion from A_1^ to B_1^, then the implicit handle operator for B_1 is applicable.
- Otherwise, if there is a handle conversion from B_1^ to A_1^, then the implicit handle operator for A_1 is applicable.
- Otherwise, no implicit handle equality operator is applicable.

If the operands to an *equality-expression* are not handles, no implicit handle equality operator is applicable.

[*Note*: The rules here have the following implications:

- The implicit handle equality operators cannot be used to compare types that are known to be different. For example, two types A and B that derive from System::Object could never be successfully compared for identify. Similarly, if A is a ref class and B is an interface that A does not implement, then no implicit handle equality operator applies.
- The implicit handle equality operators do not permit value class operands to be campared without a user-defined equality operator.
- The implicit handle equality operators never cause boxing conversions to occur for an operand. Such a conversion would be meaningless.

end note]

When overload resolution rules select an equality operator other than the implicit handle equality operator, selection of an implicit handle equality operator can be forced by explicitly casting one or both operands to `System::Object^`.

15.8.2 Delegate equality operators

Every delegate type implicitly provides the following predefined comparison operators:

```
bool operator ==(Delegate^ x, Delegate^ y);
bool operator !=(Delegate^ x, Delegate^ y);
```

These are implemented in terms of `System::Delegate::Equals`. If the two operands are of different delegate types, the expression is ill-formed. [*Rationale*: Two different delegate types can never successfully result in equality. Overload resolution can promote both delegate types to System::Delegate postponing equality failure to run-time. *end rationale*]

15.8.3 String equality

Equality of `System::String` handles is defined by `System::String::operator==` and `System::String::operator!=`.

15.9 Logical AND operator

The C++ Standard (§5.14/1) is augmented as follows:

> The && operator groups left-to-right. The operands are both implicitly converted to type `bool` (clause 4). If that conversion is ill-formed and the operand is a handle type or a type given by a generic type parameter not constrained by the value class constraint, the operand is tested for the null value, returning `true` if not null and `false` if it is null. Otherwise, if the conversion to `bool` is ill-formed and the operand is not a handle type or a type given by a generic type parameter not constrained by the value class constraint, the program is ill-formed. The result is `true` if both operands are `true` and `false` otherwise. Unlike &, && guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is `false`.

15.10 Logical OR operator

The C++ Standard (§5.15/1) is augmented as follows:

> The || operator groups left-to-right. The operands are both implicitly converted to `bool` (clause 4). If that conversion is ill-formed and the operand is a handle type or a type given by a generic type parameter not constrained by the value class constraint, the operand is tested for the null value, returning `true` if not null and `false` if it is null. Otherwise, if the conversion to `bool` is ill-formed and the operand is not a handle type or a type given by a generic type parameter not constrained by the value class constraint, the program is ill-formed. It returns `true` if either of its operands is `true` and `false` otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to `true`.

15.11 Conditional operator

With regard to expressions of the following forms

```
e ? p : nullptr
e ? nullptr : p
e ? h : nullptr
e ? nullptr : h
```

where `e` is an expression that can be implicitly converted to `bool`, `p` has pointer type, and `h` has handle type, the C++ Standard (§5.16/6) is augmented to

> The second and third operands have pointer type, or one has pointer type and the other is a null pointer constant or null value constant; pointer conversions and qualification conversions are performed to bring them to their composite pointer type. The result is of the composite pointer type. If either the second or the third operands have a handle type, and the other operand is the null value constant, the result is of the handle type.

15.12 Assignment operators

In the expression `E1` *op*= `E2`, `E1` can be a property, because after synthesis that expression is treated as `E1 = E1` *op* `E2`.

A program that attempts to use the result of an assignment expression of the form `E1 = E2` in which `E1` is a `property`, is ill-formed. [*Note*: The type of the result of such an expression is the type of `E1`, and since the set accessor function for the property has type `void`, the result has type `void`. *end note*]

For information about the synthesis of compound assignment operators see (§19.7.4). Property and event rewrite rules are covered in §15.14.

The left operand of an assignment shall be an lvalue or a gc-lvalue.

15.13 Constant expressions

The C++ Standard (§5.19/2) provides a list of "Other expressions [that] are considered *constant-expression*s only for the purpose of non-local static object initialization." That list is augmented by the addition of the following:

- the null value constant.

A literal field can be used in any context that permits a literal of the same type. As such, a literal field can be present in a compile-time constant expression.

To accommodate the addition of literal fields, the C++ Standard is augmented by the addition of the following after §5.19/3:

> A literal constant expression includes arithmetic constant expression, string literals of type `System::String^`, and the null value constant `nullptr`.

String concatenation expressions that use only literal values can be evalutated by the compiler and are therefore considered compile-time expressions. [*Example*:

```
#define X 42

ref struct R {
  literal String^ Truth = "The meaning of life is " + X;
};
```

end example]

When a static const variable is brought into scope through `#using`, the compiler cannot treat it as a literal value. Thus, it cannot be used in contexts in which a literal is needed (such as a template non-type argument or native array size). However, when a static const variable is brought in via `#include`, the Standard C++ rules as to whether it can be used as a literal, are followed.

15.14 Property and event rewrite rules

For the purposes of lookup, properties are treated as class data members. The evaluation of an expression involving one or more properties requires that expression to be rewritten using the accessor functions (§19.5.3) for those properties.

Before a property expression is rewritten using accessor functions, operator synthesis rules (§19.7.4) shall be applied to that expression. (As a result, the property rewrite process will never encounter a compound assignment operator.)

Consider the expression `E1` @ `E2`, in which @ represents a binary operator. If `E2` is a property, it shall be rewritten as a call to that property's get accessor function, before further evaluation. If `E1` is a property, then if @ is the simple assignment operator, the expression shall be rewritten as a call to the property's set accessor function; otherwise, `E1` shall be rewritten as a call to the property's get accessor function..

If the expression `E` evaluates to a property and `E` is not an operand to a binary operator, `E` shall be rewritten as a call to that property's get accessor function.

Rewrites for property expressions are different for scalar and indexed properties. If `P` is a scalar property (§19.5):

- The property get rewrite shall be `P::get()`.
- The property set rewrite shall be `P::set(`*expression*`)`, where *expression* corresponds to the right-hand side of a simple assignment operator expression.

If `E` is an indexed property (§19.5), it has the general form `P[`*expression-list*`]`.

- The property get rewrite shall be `P::get(`*expression-list*`)`.
- The property set rewrite shall be `P::set(`*expression-list*`,` *expression*`)`, where *expression* corresponds to the right-hand side of a simple assignment operator expression.

[*Example*: Given that `P`, `Q`, and `R` are scalar properties, the expression

```
P += Q * !R
```

is converted by operator synthesis to

```
P = P + Q * !R
```

which is then rewritten as

```
P::set(P::get() + Q::get() * !R::get())
```

In addition, given that `A`, `B`, and `C` are indexed properties, the expression

```
A[i] = B[j,k] + C[l,m,n]
```

is rewritten as

```
A::set(i, B::get(j,k) + C::get(l,m,n))
```

end example]

The rewrite rules for the prefix and postfix `++` and `--` operators are discussed in §19.7.3.

If lookup finds multiple properties by the same name in a class, an expression of the form `P`[*expression-list*] shall always be interpreted as an indexed property access (even if the number of arguments does not match any existing property). If the only property found is a scalar property, the rewrite rule used shall be that for a scalar property get, and the subscript operator shall be applied to the result of that property get.

[*Example*: In the following example, the class `R` has only one property by the name `P`. Since it is a scalar property, the subscript operator is applied to the result of the property.

```
ref struct R {
   property String^ P { String^ get() { … } }
};

int main() {
   R^ r = gcnew R;
   wchar_t c = r->P[0];    // calls String's default-indexed property
}
```

In the next example, `R` has two properties by the name `X`. Thus, all subscripts to `X` are interpreted as indexed properties. Because no set function exists that matches the overload of the rewrite, the following code is ill-formed.

```
ref class R {
  array<int>^ MyArray;

public:
  R() { MyArray = gcnew array<int>(10); }

  property array<int>^ X {
    array<int>^ get() { return MyArray; }
  }

  property int X[int] {
    int get(int i) { return i*i; }
  }
};

int main() {
  R r;
  r.X[2] = 1;     // error - no R::X::set(int,int) exists
  int y = r.X[2]; // calls R::X::get(int)
}
```

end example]

After property expressions are rewritten, the resulting expression is reevaluated using existing rules. At that time, it is possible that overload resolution will fail to find an acceptable function, in which case, the program is ill-formed. [*Example*: An indexed property is rewritten yet no property access method takes the

required number of arguments. If a property only has a get accessor function, yet an expression involving that property is rewritten as a property set, lookup will fail to find a set accessor function. *end example*]

Before being rewritten, properties act like fields. As such, when lookup finds a property or field name, it does not look further in the base classes for more property names, even if the class is a hidebysig class (§10.7). However, after being rewritten, the accessor functions for a property do follow the same rules as other functions for hidebysig lookup.

When the left operand of a compound assignment operator is an event, operator synthesis shall not be applied.

Given the expression `E1` @ `E2`, in which @ represents a binary operator, if `E1` is an event, the event is rewritten with the following rules:

- If @ is `+=`, the expression is rewritten as an event add, `E1::add(E2)`.
- If @ is `-=`, the expression is rewritten as an event remove, `E1::remove(E2)`.

Otherwise, the program is ill-formed.

Given the expression `E(`*expression-list*`)`, if `E` is an event, the expression is rewritten as an event raise, `E::raise(`*expression-list*`)`.

All other usages of an event in an expression are ill-formed.

[*Example*: Given that `V` is an event and `D` is a delegate, the expression `V += D` is rewritten as `V::add(D)`, the expression `V -= D` is rewritten as `V::remove(D)`, the expression `V(this, e)` is rewritten as `V::raise(this, e)`. *end example*]

After an event expression is rewritten, it is reevaluated using existing rules. At that time, it is possible that overload resolution will fail to find an acceptable function, in which case, the program is ill-formed. [*Example*: A delegate cannot be added to an event if they have different delegate types. *end example*]

16. Statements

Unless stated otherwise in this clause, all existing statements are supported and behave as specified in the C++ Standard (§6).

16.1 Selection statements

16.1.1 The switch statement

A program is ill-formed if it uses a `switch` statement to transfer control in to a *finally-clause*.

16.2 Iteration statements

In addition to the three iteration statements specified by Standard C++ (§6.5), the *iteration-statement* production is augmented to include the `for each` statement.

iteration-statement:
`while` `(` *condition* `)` *statement*
`do` *statement* *while* `(` *expression* `)` `;`
`for` `(` *for-init-statement* *condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
`for each` `(` *type-specifier-seq* *declarator* `in` *assignment-expression* `)` *statement*

16.2.1 The for each statement

The `for each` statement enumerates the elements of a collection, executing the *statement* for each element of that collection.

Together, the *type-specifier-seq* and *declarator* declare the ***iteration variable*** of the statement. This iteration variable corresponds to a local variable with a scope that extends over *statement*. During execution of a `for each` statement, the iteration variable represents the collection element for which an iteration is currently being performed.

The type of *assignment-expression* shall be a collection type (as defined below), and it shall be possible to convert from the element type of the collection to the type of the iteration variable using `safe_cast`. If *assignment-expression* has the value `nullptr`, a `System::NullReferenceException` is thrown.

A type is said to be a ***collection type*** if it implements the `System::Collections::IEnumerable` interface, or implements `System::Collections::Generic::IEnumerable` interface, or implements the ***collection pattern*** by meeting all of the following criteria:

Expression	Return Type	Assertion/NotePre/Post-Condition
`e = c.GetEnumerator()`	`E`	`E` is the enumerator type.
`e = c->GetEnumerator()`		
`e.MoveNext()`	A value that can be used as a condition (see §14.2.1)	True if the current instance was successfully advanced to the next element; false if the current instance has passed the end of the collection.
`e->MoveNext()`		
`e.Current`	rvalue, lvalue, or gc-lvalue that is an element of the collection	This is the ***element type*** of the collection type.
`e->Current`		

where `c` is a collection of object convertible to type `T`, and `e` is an enumerator that can be used for iteration over a collection.

A type that implements `IEnumerable` is also a collection type, even if it doesn't satisfy the conditions above. (This is possible if it implements `IEnumerable` via explicit interface member implementations.)

The `System::Array` type (§24.1.1) is a collection type, and since all CLI array types derive from `System::Array`, any CLI array type expression is permitted in a `for each` statement. For single-dimensional CLI arrays, the `for each` statement enumerators traverses the CLI array elements in increasing order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional CLI arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

A `for each` statement of the form

```
for each (T d in <collection-expr>) statement
```

in which <*collection-expr*> is a collection of `T`, is executed as if it were written as follows if `GetEnumerator` returns a handle:

```
{
    <enumeration-type>^ e;
    try {
        e = <collection-expr>.GetEnumerator();
        while(e->MoveNext())
            T d = safe_cast<T>(e->Current);
            statement
        }
    } finally {
        delete e;
    }
}
```

where `e` is a non-user-accessible temporary and <*enumeration-type*> is the type of the object returned by the `GetEnumerator` function. If `GetEnumerator` returns a pointer, the execution is the same as the handle case except `e` is declared as a pointer. If `GetEnumerator` does not return a pointer or handle, the statement is executed as if it were writtern as follows:

```
{
    <enumeration-type> e = <collection-expr>.GetEnumerator();
    while(e.MoveNext())
        T d = safe_cast<T>(e.Current);
        statement
    }
}
```

[*Example*: The following program pushes the values 0 through 9 onto an integer stack and then uses a `for each` loop to display the values in top-to-bottom order.

```
int main() {
   Stack<int>^ s = gcnew Stack<int>;
   for (int i = 0; i < 10; ++i)
      s->Push(i);
   for each (int i in s)
      Console::Write("{0} ", i);
   Console::WriteLine();
}
```

The output produced is:

```
9 8 7 6 5 4 3 2 1 0
```

A CLI array is an instance of a collection type, so it too can be used with `for each`:

```
int main() {
   array<double>^ values = {1.2, 2.3, 3.4, 4.5};
   for each (double value in values)
      Console::WriteLine(value);
}
```

The output produced is:

```
1.2 2.3 3.4 4.5
```

end example]

16.3 Jump statements

16.3.1 The break statement

A program is ill-formed if it uses a `break` statement to transfer control out of a *finally-clause*.

16.3.2 The continue statement

A program is ill-formed if it uses a `continue` statement to transfer control out of a *finally-clause*.

16.3.3 The return statement

A program is ill-formed if it has a `return` statement in a *finally-clause*.

16.3.4 The goto statement

A program is ill-formed if it uses a `goto` statement to transfer control in to or out of a *finally-clause*.

16.4 The try block

In the grammar specified by Standard C++ (§15), the *try-block* and *function-try-block* productions are augmented to include an optional *finally-clause*, as follows:

try-block:
`try` *compound-statement* *handler-seq*
`try` *compound-statement* *finally-clause*
`try` *compound-statement* *handler-seq* *finally-clause*

function-try-block:
`try` *ctor-initializer*$_{opt}$ *function-body* *handler-seq*
`try` *ctor-initializer*$_{opt}$ *function-body* *finally-clause*
`try` *ctor-initializer*$_{opt}$ *function-body* *handler-seq* *finally-clause*

finally-clause:
`finally` *compound-statement*

The statements in a *finally-clause* are always executed when control leaves the associated *try-block*'s or *function-try-block*'s *compound-statement*. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a `break`, `continue`, `goto`, or `return` statement, or as a result of propagating an exception out of that *try-block*'s or *function-try-block*'s *compound-statement*.

If an exception is thrown during execution of the statements in a *finally-clause*, the exception is propagated to the next enclosing *try-block* or *function-try-block*. If another exception was in the process of being propagated, that exception is lost.

[*Example*:

```
class MyException {};
void f1();
void f2();

int main() {
   try {
      f1();
   }
   catch (const MyException& re) {
      …
   }
}
```

```
void f1() {
   try {
      f2();
   }
   finally {
      …
   }
}
void f2() {
   if ( … ) throw MyException();
}
```

If the call to `f2` returns normally, the finally block is executed after `f1`'s try block terminates. If the call to `f2` results in an exception, the finally block is executed before `main`'s catch block gets control. *end example*]

[*Note*: A program is ill-formed if it:

- uses a `break` or `continue`, or `goto` statement to transfer control out of a *finally-clause*.
- has a `return` statement in a *finally-clause*.
- uses `goto` or `switch` statement to transfer control into a *finally-clause*.

end note]

17. Namespaces

C++/CLI has no additional namespace features beyond those provided by Standard C++.

17.1 Reserved namespaces

The namespace `cli` is reserved. The only elements permitted in this namespace shall be those defined by the language specification. [*Example*: These include `array` (§24.1), `interior_ptr` (§12.3.6.1), `pin_ptr` (§12.3.7.1), and `safe_cast` (§15.3.11). *end example*] A program that attempts to add a declaration to the namespace `cli` is ill-formed.

A program can employ a *using-directive* for the namespace `cli`, or have a *using-declaration* for an entity in that namespace.

A conforming implementation shall correctly consume assemblies containing public names that start with the C++/CLI-equivalent prefix `::cli::`. [*Note*: Such names might be produced from C#, for example. *end note*]

18. Functions

18.1 <cstdarg>-style variable-argument lists

If a function whose *parameter-declaration-clause* terminates with an ellipsis, is called with `nullptr` as any argument that corresponds to the ellipsis, the program is ill-formed. [*Note*: The type of `nullptr` is not directly expressible in the language, yet the `<cstdarg>` machinery requires expressible types, so it can extract the arguments from the variable-argument list passed. *end note*] [*Example*:

```
void f(const char* pc, ...) {}

int main() {
   f(nullptr);             // valid
   f("abc", nullptr);      // ill-formed
   f("abc", 10, nullptr);  // ill-formed
}
```

end example]

18.2 Name lookup

For metadata details, see §34.6.1.

18.3 Overload resolution

To accommodate string literal conversion, boxing conversion, Boolean, and handle conversion, Table 9, "conversions", in the C++ Standard, §13.3.3.1.1, "Standard conversion sequences", is augmented by the addition of some new rows, as indicated by shading below:

Conversion	Category	Rank	Subclause
No conversion required	Identity	Exact Match	
String literal conversion			
Lvalue-to-rvalue conversion	Lvalue Transformation		4.1
Array-to-pointer conversion			4.2
Function-to-pointer conversion			4.3
Qualification conversions	Qualification Adjustment		4.4
Boolean equivalence			
Integral promotions	Promotion	Promotion	4.5
Floating point promotion			4.6
Boxing conversion			
Integral conversions	Conversion	Conversion	4.7
Floating point conversions			4.8
Floating-integral conversions			4.9
Pointer conversions			4.10
Pointer to member conversions			4.11
Handle conversions			
Boolean conversions			4.12

18.4 Parameter arrays

Standard C++ supports variable-length argument lists for both member and non-member functions; however, the approach used is not type-safe. C++/CLI adds a type-safe way using ***parameter arrays***. A parameter array is defined as follows:

parameter-array:
attributes$_{opt}$ `...` *parameter-declaration*

A *parameter-array* consists of an optional set of *attributes* (§29), an ellipsis punctuator, and a *parameter-declaration*. A parameter array declares a single parameter of the given CLI array type. The CLI array type of a parameter array shall be a single-dimensional CLI array type (§24.1). In a function invocation, a parameter array permits either a single argument of the given CLI array type to be specified, or it permits zero or more arguments of the CLI array element type to be specified. The program is ill-formed if the *parameter-declaration* contains a default argument. [*Example*:

```
void f(... array<Object^>^ p);

int main() {
   f();
   f(nullptr);
   f(1, 2);
   f(nullptr, nullptr);
   f(gcnew array<Object^>(1));
   f(gcnew array<Object^>(1), gcnew array<Object^>(2));
}
```

end example]

[*Example:*

```
void F1(... array<String^>^ list) {
   for (int i = 0 ; i < list->Length ; i++ )
      Console::Write("{0} ", list[i]);
   Console::WriteLine();
}

void F2(... array<Object^>^ list) {
   for each (Object^ element in list)
      Console::Write("{0} ", element);
   Console::WriteLine();
}

int main() {
   F1("1", "2", "3");
   F2(1, L'a', "test");
   array<String^>^ myarray
      = gcnew array<String^> {"a", "b", "c" };
   F1(myarray);
}
```

The output produced is as follows:

```
1 2 3
1 a test
a b c
```

end example]

When a function with a parameter array is invoked, the invocation is processed as if a *new-expression* (§15.4.6) with an *array-init* (§24.6) was inserted around the list of arguments corresponding to the parameter array.

When there are zero arguments given for the parameter array, a zero-length CLI array shall be passed.

[*Example:* Given the declaration

```
void F(int x, int y, ... array<Object^>^ args);
```

the following invocations of the function

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

correspond exactly to

```
F(10, 20, nullptr);
F(10, 20, gcnew array<System::Object^> {30, 40});
F(10, 20, gcnew array<System::Object^> {1, "hello", 3.0});
```

end example]

Parameter array parameters can be passed to functions that take non-parameter CLI array arguments of the corresponding type. [*Example:*

```
void f(array<int>^ pArray);   // not a parameter array
void g(double value, ... array<int>^ p) {
   f(p);                      // Ok
}
```

end example]

An argument of type `array` can be passed to a function having a parameter array parameter, without invoking a parameter array conversion sequence. [*Note*: An `array` argument that can be converted to the parameter array's type without a parameter array conversion, as happens in a handle conversion, will not prefer the parameter array conversion sequence. *end note*]

When a function with a parameter array is included in the candidate set for overload resolution, two function signatures are included. Given a function signature T_R `F(`T_1`, `T_2`, …, ... array<`T_P`>)`, the ***exact form*** replaces the parameter array parameter with a normal array parameter (T_R `F(`T_1`, `T_2`, …, array<`T_P`>)`), and the ***expanded form*** replaces the parameter array parameter with a series of parameters of the array's element type (T_R `F(`T_1`, `T_2`, …, `T_{P1}`, `T_{P2}`, …, `T_{PN}`)`). The number of parameters in the exanded form matches the number of arguments to the function invocation. Both signatures are included before the elimination of viable functions. If the expanded form is selected by overload resolution, a parameter array conversion sequence is used to call the function.

For metadata details, see §34.6.2.

18.5 Importing native functions

Functions defined in native code in one assembly can be invoked from another assembly by using the `DllImportAttribute` (from namespace `System::Runtime::InteropServices`) on the declaration of a global or namespace scope function declaration or on a static member function of a ref class or value class. Such function declarations shall not also be definitions. This attribute shall not be applied to an instance member function. This attribute provides the name of the native code assembly, the name of the function within that assembly, the calling convention to be used to call the native code function, and the character set used for string marshaling. [*Example*:

```
// MyCLib.h
using namespace System::Runtime::InteropServices;
[DllImport("MyCLib.dll", CallingConvention =
CallingConvention::StdCall, EntryPoint="Hypot" )]
extern "C" double Hypotenuse(double s1, double s2);
```

```
// MyCLibApp.cpp
#include "MyCLib.h"

int main() {
   Console::WriteLine("Hypotenuse = {0}", Hypotenuse(3, 4));
}
```

In this case, the function named `Hypot` resides in the shared library MyCLib.dll. This name is mapped to that of the program element to which the attribute is applied; namely, to `Hypotenuse`. A calling convention is specified, as appropriate.

The way in which the `Hypot` function is written, is implementation-defined. Here is a version written for one implementation:

```
// MyCLib.c
#include <math.h>
__declspec(dllexport) double __stdcall Hypot(double side1, double side2)
{
   return sqrt((side1 * side1) + (side2 * side2));
}
```

In the following example, the Standard C library function `strcmp` is imported and `String^`-to-`char*` conversion occurs on the arguments by virtue of the `MarshalAsAttribute` attribute (from namespace `System::Runtime::InteropServices`):

```
using namespace System::Runtime::InteropServices;
[DllImport("msvcrt.dll", CallingConvention = CallingConvention::Cdecl)]
extern "C" int strcmp([MarshalAs(UnmanagedType::LPStr)]
   System::String^ s1,
   [MarshalAs(UnmanagedType::LPStr)] System::String^ s2);

int main() {
   String^ str1 = "red";
   String^ str2 = "RED";
   Console::WriteLine("Compare: {0}", strcmp(str1, str2));
}
```

end example]

For metadata details, see §34.6.3.

18.6 Non-member functions

[*Note*: Non-member functions are treated by the CLI as members of some unspecified class; however, in C++/CLI source code, such functions cannot be qualified explicitly with that class name. *end note*]

For metadata details, see §34.6.4.

18.7 Attributes

*function-definition*s (§19.4) and function declarations resulting from either a *simple-declaration* or the first production of *member-declaration* can have attributes.

The simple-declaration production is augmented as follows to allow attributes on function declarations and global variables:

simple-declaration:
attributes$_{opt}$ *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$;

19. Classes and members

This clause specifies the features of a class that are new in C++/CLI. However, not all of these features are available to all classes. The class-related features that are supported by native classes (§20), ref classes (§21), value classes (§22), and interfaces (§25), are specified in the clauses that define those types. [*Note:* A summary of that support is shown in the following table:

Feature	Native class	Ref class	Value class	Interface
Assignment operator	X	X		
Class modifier	X	X	X	
Copy constructor	X	X		
Default constructor	X	X		
Delegate definitions	X	X	X	X
Destructor	X	X		X
Events		X	X	X
Finalizer		X		
Function modifiers	X	X	X	n/a
Initonly field		X	X	X
Literal field		X	X	X
Member of delegate type		X	X	
Override specifier	X	X	X	n/a
Parameter arrays	X	X	X	X
Properties		X	X	X
Reserved member names		X	X	X
Static constructor		X	X	X
Static operators	X	X	X	X

end note]

19.1 Class definitions

In the C++ Standard (§9), a *class-specifier* is used to define a class. This grammar is augmented to accommodate the addition of public and private classes, as follows:

class-specifier:
attributes$_{opt}$ *top-level-visibility*$_{opt}$ *class-head* { *member-specification*$_{opt}$ }

attributes is described in §29, *top-level-visibility* is described in §12.4.

class-head (§9) is augmented to support class modifiers (§19.1.1):

class-head:
class-key *identifier*$_{opt}$ *class-modifiers*$_{opt}$ *base-clause*$_{opt}$
class-key *nested-name-specifier* *identifier* *class-modifiers*$_{opt}$ *base-clause*$_{opt}$
class-key *nested-name-specifier*$_{opt}$ *template-id* *class-modifiers*$_{opt}$ *base-clause*$_{opt}$

class-key (§9) is augmented to support ref classes (§21), value classes (§22), and interface classes (§25):

class-key:
    `class`
    `struct`
    `union`
    `ref class`
    `ref struct`
    `value class`
    `value struct`
    `interface class`
    `interface struct`

To accommodate the addition of initonly and literal fields, delegates, events, generics, and properties, the syntactic class *member-declaration* in the C++ Standard (§9.2) is augmented, as follows:

member-declaration:
    attributes$_{opt}$ *initonly-or-literal*$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$ `;`
    function-definition `;`$_{opt}$
    `::`$_{opt}$ *nested-name-specifier* *template*$_{opt}$ *unqualified-id* `;`
    using-declaration
    template-declaration
    generic-declaration
    delegate-specifier
    event-definition
    property-definition

initonly-or-literal:
    `initonly`
    `literal`

Attributes are described in §29, initonly fields in §19.12, literal fields in §19.11, generics in §31, delegates in §27, events in §19.6, and properties in §19.5.

For metadata details, see §34.7.1.

19.1.1 Class modifiers

To accommodate the addition of sealed and abstract classes, the grammar for *class-head* in the C++ Standard (§9) is augmented to include an optional sequence of class modifiers, as follows:

class-modifiers:
    class-modifiers$_{opt}$ *class-modifier*

class-modifier:
    `abstract`
    `sealed`

If the same modifier appears multiple times in a *class-modifiers*, the program is ill-formed.

[*Note:* `abstract` and `sealed` can be used together; that is, they are not mutually exclusive. As non-member functions are not CLS-compliant, a substitute is to use an abstract sealed class, which can contain static member functions. This is the utility class pattern. *end note*]

A class that is both `abstract` and `sealed` shall not have a *base-clause*, instance constructors, or instance members; it shall have only static members, nested types, literal fields, and typedefs.

The `abstract` and `sealed` modifiers are discussed in §19.1.1.1 and §19.1.1.2, respectively.

19.1.1.1 Abstract classes

An abstract class follows the rules of Standard C++ for abstract classes (§10.4); however, a class definition containing the `abstract` class modifier need not contain any abstract functions. [*Example:*

```
struct B abstract {
   void f() { }
};

struct D : B { };

int main() {
   B b;              // error: B is abstract
   D d;              // ok
}
```

end example]

A ref class that contains any abstract functions (including accessor functions) shall be explicitly declared `abstract`.

For metadata details, see §34.7.1.1.

19.1.1.2 Sealed classes

The `sealed` modifier is used to prevent derivation from a class. The program is ill-formed if a sealed class is specified as the base class of another class. [*Example:*

```
struct B sealed {
};

struct D : B {       // error, cannot derive from a sealed class
};
```

end example]

Whether or not a class is sealed has no effect on whether or not any of its member functions are, themselves, sealed.

[*Note:* The `sealed` modifier is primarily used to prevent unintended derivation, but it also enables certain runtime optimizations. In particular, because a sealed class is known never to have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations. *end note*]

For metadata details, see §34.7.1.2.

19.2 Reserved member names

To facilitate the underlying C++/CLI runtime implementation, for each CLI class type member definition that is a property or event, the implementation shall reserve several names based on the kind of the member definition (§19.2.1, §19.2.2). A program is ill-formed if it contains a class that declares a property or event, and a member whose name matches any of that property or event's reserved names.

During lookup, the reserved names are invisible.

[*Note:* The reservation of these names serves several purposes:

- To allow other languages to interoperate using an ordinary identifier as a function name for get or set access.
- Partition I of the CLI standard requires these names for CLS-producer languages.

end note]

In order to accommodate the CLI notion of finalizers, several names are reserved in CLI class types for functions (§19.2.3).

19.2.1 Member names reserved for properties

For a scalar or named indexed property `P` (§19.5), the following names are reserved:

```
get_P
set_P
```

Both names are reserved, even if the scalar or named indexed property is read-only or write-only.

[*Example:*

```
ref struct A {
   property int P {
      int get() { return 123; }
   }
};

ref struct B : A {
   int get_P() {        // error
      return 456;
   }
};
```

end example]

For a CLI class that has a default-indexed property (§19.5), the following names are reserved:

```
get_Item
set_Item
```

Both names are reserved, even if the default-indexed property is read-only or write-only.

The default name suffix, `Item`, of a default-indexed property can be changed by applying the `DefaultMemberAttribute` (from namespace `System::Reflection`) to that property's parent type. All default-indexed properties in a class shall have the same underlying name. Once a default-indexed property's name has been changed in this way, it shall not be changed in any class derived from that property's parent type. If two interface classes declare a default-indexed property, and each specifies a different name via this attribute, a program is ill-formed if it declares a type that implements both interfaces.

Alternatively, the program can change the default name suffix by applying the `System::Runtime::CompilerServices::IndexerNameAttribute` to all default-indexed properties within a class. The resulting metadata will replace `IndexerNameAttribute` with `DefaultMemberAttribute` (see §34.7.5). A program is ill-formed if it uses both the `IndexerNameAttribute` and `DefaultMemberAttribute` to specify the default name suffix for the same member. Similarly, a program is ill-formed if two default-indexed properties in the same class use `IndexerNameAttribute` to specify different underlying names; all default-indexed properties in a class shall have the same `IndexerNameAttribute` applied. [*Rationale*: C++/CLI supports `IndexerNameAttribute` because that is the approach used by several other languages, and it supports `DefaultMemberAttribute` because that is what is actually emitted in metadata. *end rationale*]

For metadata details, see §34.7.5.

19.2.2 Member names reserved for events

For an event `E` (§19.6), the following names are reserved:

```
add_E
remove_E
raise_E
```

19.2.3 Member names reserved for functions

For CLI class types, the following function name and parameter list combinations are reserved (where `T` is any ref class name):

```
Dispose()
Dispose(bool)
Finalize()
__identifier("~T")()
__identifier("!T")()
```

19.2.4 Possible collision with reserved property and event names

The reserved name patterns for any given property or event are reserved only in the class defining that property or event.

[*Note*: The program

```
ref struct B {
   int get_X() { Console::WriteLine("B::get_X"); return 1; }
};

ref struct D : B {
   property int X {
      int get() { Console::WriteLine("D::X::get"); return 2; }
   }
};

int main() {
  D d;
  d.get_X();
}
```

prints "`B::get_X`".

If a property or event is virtual and no base class has a virtual property or event of the same name, the underlying accessor functions generated for the property are introducing functions. That is, they will not override functions from the base class. The program

```
ref struct B {
   virtual int get_X() { Console::WriteLine("B::get_X"); return 1; }
};

ref struct D : B {
   virtual property int X {
      int get() { Console::WriteLine("D::X::get"); return 2; }
   }
};

int main() {
  D d;
  d.get_X();
}
```

prints "`B::get_X`". The only way to override `B::get_X` when deriving from `D` is to use a named override. *end note*]

If a function other than a property or event accessor in a derived class overrides a virtual accessor function from the base class, the program is ill-formed. These functions shall be marked with the `new` function modifier. This is true even if the name of the accessor function in the base class does not use the canonical `get_X`, `set_X`, `add_X`, `remove_X`, or `raise_X` names (which can only happen when `#using` an assembly that was generated in a language other than C++/CLI). [*Example*:

```
ref struct B {
   virtual property int X {
      int get() { Console::WriteLine("B::X::get"); return 1; }
   }
};

ref struct D : B {
   virtual int get_X() new { Console::WriteLine("D::get_X"); return 2; }
};

int main() {
  D d;
  d.get_X();
}
```

Without the `new` function modifier applied to `D::get_X`, the program is ill-formed. *end example*]

19.3 Data members

A ref or value class type can have the attribute `StructLayoutAttribute` (in namespace `System::Runtime::InteropServices`). This attribute can be used to specify the layout of a data structure, the alignment, the size, and the marshalling of strings. An instance data member can have the attribute `FieldOffsetAttribute` (in namespace `System::Runtime::InteropServices`), which controls the exact placement of that member. (For more information on this attribute, refer to the CLI Standard.) [*Example*:

```
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Explicit)]
public value class S1 {
   [FieldOffset(0)] int v;
   [FieldOffset(4)] unsigned char c;
   [FieldOffset(8)] int w;
};

[StructLayout(LayoutKind::Sequential, Pack=4)]
public value class S2 {
   int v;
   unsigned char c;
   int w;
};

[StructLayout(LayoutKind::Explicit, Size=12, CharSet=CharSet::Unicode)]
public ref class S3 {
   [FieldOffset(0)] int* pi;
   [FieldOffset(0)] unsigned int ptrValue;
};
// S3 is intended to behave like a union and should be treated as such
```

end example]

Data members can have applied to them the attribute `MarshalAsAttribute` (in namespace `System::Runtime::InteropServices`). For more information on this attribute, see §18.5.

For metadata details, see §34.7.3.

19.4 Functions

To allow attributes on a function definition, the Standard C++ grammar for *function-definition* (§8.4) is augmented, as follows:

function-definition:
attributes$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* *function-modifiers*$_{opt}$ *override-specifier*$_{op}$ *ctor-initializer*$_{opt}$ *function-body*
attributes$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* *function-modifiers*$_{opt}$ *override-specifier*$_{opt}$ *function-try-block*

The addition of overriding specifiers and function modifiers requires augmentations to the Standard C++ grammar for *function-definition* and to one of the productions of *member-declarator*. [*Note:* The two new optional syntax productions, *function-modifier* and *override-specifier*, appear in that order, after *exception-specification*, but before *function-body* or *function-try-block*. *end note*]

To allow attributes, function modifiers, and an override specifier on a function declaration that is not a definition, one of the productions for the Standard C++ grammar for *member-declarator* (§9.2) is augmented, as follows:

member-declarator:
declarator *function-modifiers*$_{opt}$ *override-specifier*$_{opt}$
declarator *constant-initializer*$_{opt}$
identifier$_{opt}$ `:` *constant-expression*

function-modifiers:
function-modifiers$_{opt}$ *function-modifier*

function-modifier:
    `abstract`
    `new`
    `override`
    `sealed`

The set of attributes on a function declaration that is not a definition shall be a subset of the set of attributes on the corresponding function definition. Attributes are described in §29.

*function-modifier*s are discussed in the following subclauses: `abstract` in §19.4.3, `new` in §19.4.4, `override` in §19.4.1, and `sealed` in §19.4.2. *override-specifier* is discussed in §19.4.1.

A member function declaration containing any of the *function-modifier*s `abstract`, `override`, or `sealed`, or an *override-specifier*, shall explicitly be declared `virtual`. [*Rationale:* A major goal of this new syntax is to let the programmer state his intent, by making overriding more explicit, and by reducing silent overriding. The `virtual` keyword is required on all virtual functions, except in the one case where backwards compatibility with Standard C++ allows the `virtual` keyword to be optional. *end rationale*]

If a function contains both `abstract` and `sealed` modifiers, or it contains both `new` and `override` modifiers, it is ill-formed.

An out-of-class member function definition shall not contain a *function-modifier* or an *override-specifier*.

If a destructor or finalizer (§19.13) contains an *override-specifier*, or a `new` or `sealed` *function-modifier*, the program is ill-formed.

The Standard C++ grammar for *parameter-declaration-clause* (§8.3.5) is augmented to include support for passing parameter arrays, as follows:

parameter-declaration-clause:
    parameter-declaration-list$_{opt}$ `...`$_{opt}$
    parameter-declaration-list `,` `...`
    parameter-array
    parameter-declaration-list `,` *parameter-array*

There shall be only one parameter array for a given function or instance constructor, and it shall always be the last parameter specified.

Parameter arrays are discussed in §18.4.

For metadata details, see §34.7.4.

19.4.1 Override functions

The Standard C++ grammar for *direct-declarator* is augmented to allow the function modifier `override` as well as override specifiers.

override-specifier:
    `=` *overridden-name-list*
    pure-specifier

overridden-name-list:
    id-expression
    overridden-name-list `,` *id-expression*

In Standard C++, given a derived class with a function that has the same name, parameter-type-list, and cv-qualification of a virtual function in a base class, the derived class function always overrides the one in the base class, even if the derived class function is not declared `virtual`. This is known as ***implicit overriding***. A program containing an implicitly overridden function in ref classes and value classes is ill-formed. [*Note:* A programmer can eliminate the diagnostic by using explicit or named overriding, as described below. *end note*]

With the addition of the function modifier `override` and override specifiers, C++/CLI provides the ability to indicate ***explicit overriding*** and ***named overriding***, respectively.

If either the *function-modifier* `override` or an *override-specifier* is present in the derived class function declaration, no implicit overriding takes place. [*Example:*

```
ref struct B {
   virtual void F() {}
   virtual void F(int i) {}
};

ref struct D1: B {
   virtual void F() override {}        // explicitly overrides B::F()
};

ref struct D2: B {
   virtual void F() override {}        // explicitly overrides B::F()
   virtual void G(int i) = B::F {}     // named override of B::F(int)
};

ref struct D3: B {
   virtual void F() new = B::F {}      // named override of B::F()
};
```

end example]

[*Note:* A member function declaration containing the *function-modifier* `override` or an *override-specifier* shall explicitly be declared `virtual` (§19.2.4). *end note*]

An *override-specifier* contains a comma-separated list of names designating the virtual functions from one or more direct or indirect base classes that are to be overridden.

An *id-expression* that designates an overridden name shall designate a single function to be overridden. Lookup for the name given in the *id-expression* starts in the containing class. [*Note*: If the *id-expression* is an unqualified name, and the containing class has a function by the same name the program is ill-formed. It is not possible to override a function within the same class. *end note*] Further qualification is necessary if the base class name is ambiguous. That function shall have the same parameter-type-list and cv-qualification as the overriding function, and the return types of the two functions shall be the same.

[*Example:*

```
interface class I {
   void F();
};

ref struct B {
   virtual void F() { … }
};

ref struct D : B, I {
   virtual void G() = B::F, I::F { … } // override B::F and I::F
};
```

Both `B::F` and `I::F` must be listed separately. If the named override used just `F`, two names are found. Named overrides must designate a single function. *end example*]

[*Note:* The same overriding behavior can sometimes be achieved in different ways. For example, given a base class `A` with a virtual function `f`, an overriding function might have an *override-specifier* of `A::f`, have no *override specifier* or `override` *function modifier*, have the *function-modifier* `override`, or a combination of the two, as in `override = A::f`. All override `A::f`. *end note*]

The name of the overriding function need not be the same as that being overridden.

A derived class shall not override the same virtual function more than once. If an implicit or explicit override does the same thing as a named override, the program is ill-formed. [*Example:*

```
interface struct I {
   void F();
};
```

```
ref struct B {
   virtual void F() { … }
   virtual void G() { … }
};

ref struct D : B, I {
   virtual void G() = B::F { … }
   virtual void F() {}  // error, would override B::F and I::F, but
                        // B::f is already overridden by G.
};
```

end example]

A class is ill-formed if it has multiple functions with the same name, parameter-type-list, and cv-qualification even if they override different inherited virtual functions. [*Example:*

```
ref struct D : B, I {
   virtual void F() = B::F { … }    // ok
   virtual void F() = I::F { … }    // error, duplicate declaration
};
```

end example]

A function can both hide and override at the same time: [*Example:*

```
interface struct I {
   void F();
};

ref struct B {
   virtual void F() { … }
};

ref struct D : B, I {
   virtual void F() new = I::F { … }
};
```

The presence of the `new` function modifier (§19.4.4) indicates that `D::F` does not override any method `F` from its base class or interface. The named override then goes on to say that `D::F` actually overrides just one function, `I::F`. *end example*]

[*Note*: An override-specifier does not introduce that name into the class. *end note*][*Example:*

```
interface struct I {
   virtual void V();
};

ref struct R {
   virtual void W() {}
};

ref struct S : R, I {
   virtual void F() = I::V, R::W {}
};

ref struct T : S {
   virtual void G() = I::V {}
   virtual void H() = R::W {}
};

void Test(S^ s) { // s could refer to an S, T, or something else
   s->W();        // ok, virtual call
   s->R::W();     // nonvirtual call to R::W
   s->S::W();     // nonvirtual call to R::W
   s->S::F();     // ok (classes derived from S might need to do this,
                  //  and there's no ambiguity in this case)
}

int main() {
   Test(gcnew S);
   Test(gcnew T);
}
```

end example]

When matching signatures for the purpose of overriding virtual functions in generic ref classes (§31.1), or implementing a function from an interface, the constraints on the type parameters are not considered. The constraints for the type parameters can differ. [*Example*: The following program

```
public interface struct P {};
public interface struct Q {};
public ref class PQ : P, Q {};

generic<typename T>
where T : P
public ref struct B {
   virtual void F(T) { Console::WriteLine("B::F"); }
};

generic<typename T>
where T : P, Q
public ref struct D : B<T> {
   virtual void F(T) override { Console::WriteLine("D::F"); }
};

int main() {
   B<PQ^>^ b = gcnew D<PQ^>;
   b->F(gcnew PQ);
}
```

prints "`D::F`". Because `D<T>^` has a handle conversion to `B<T>^` only if `T` is the same, it is not type safe when the overriding virtual function has covariant parameters to the function it is overriding (it's only type safe to override with contravariant parameters), as the parameters will be the same. *end example*]

For metadata details, see §34.7.4.1.

19.4.2 Sealed function modifier

A virtual member function marked with the *function-modifier* `sealed` cannot be overridden in a derived class. [*Example:*

```
ref struct B {
   virtual int f() sealed;
   virtual int g() sealed;
};

ref struct D : B {
   virtual int f();     // error: cannot override a sealed function
   virtual int g() new; // okay: does not override B::g
};
```

end example]

[*Note:* A member function declaration containing the *function-modifier* `sealed` shall explicitly be declared `virtual`. *end note*] If there is no `virtual` function to implicitly override in the base class, the derived class introduces the virtual function and seals it.

Whether or not any member functions of a class are sealed has no effect on whether or not that class itself is sealed.

An implicit, explicit, or (in a CLI class type, a) named override can succeed as long as there is a non-sealed virtual function in at least one of the bases. [*Example:* Consider the case in which `A::f` is sealed, but `B::f` is not. If `C` inherits from `A` and `B`, and tries to implement `f`, it will succeed, but will only override `B::f`. *end example*]

For metadata details, see §34.7.4.2.

19.4.3 Abstract function modifier

Standard C++ permits virtual member functions to be declared abstract by using a *pure-specifier*. C++/CLI provides an alternate approach via the *function-modifier* `abstract`. The two approaches are equivalent;

using both together is well-formed, but redundant. [*Example:* A class `shape` can declare an abstract function `draw` in any of the following ways:

```
virtual void draw() = 0;            // Standard C++ style
virtual void draw() abstract;       // function-modifier style
virtual void draw() abstract = 0;   // okay, but redundant
```

end example]

[*Note:* A member function declaration containing the *function-modifier* `abstract` shall be declared `virtual`. *end note*]

For metadata details, see §34.7.4.3.

For metadata implications on the parent class for both abstract functions, see §34.7.1.1.

19.4.4 New function modifier

A function need not be declared `virtual` to have the `new` function modifier. If a function is declared `virtual` and has the `new` function modifier, that function does not override another function. However, for CLI class types, it can override another function with a named override. A function that is not declared `virtual` and is marked with the `new` function modifier does not become virtual and does not implicitly override any function.

[*Example:*

```
ref struct B {
   virtual void F() { Console::WriteLine("B::F"); }
   virtual void G() { Console::WriteLine("B::G"); }
};

ref struct D : B {
   virtual void F() new { Console::WriteLine("D::F"); }
};

int main() {
   B^ b = gcnew D;
   b->F();
   b->G();
}
```

The output produced is

```
B::F
B::G
```

In the following example, hiding and overriding occur together:

```
ref struct B {
   virtual void F() {}
};

interface class I {
   void F();
};

ref struct D : B, I {
   virtual void F() new = I::F {}
};
```

The presence of the `new` function modifier indicates that `D::F` does not override any method `F` from its base classes. The named override (§19.4.1) then goes on to say that `D::F` actually overrides just one function, `I::F`. The net result is that `I::F` is overridden, but `B::F` is not.

end example]

Static functions can use the `new` modifier to hide an inherited member. [*Example*:

```
ref class B {
public:
   virtual void F() { … }
};
```

```
ref class D : B {
public:
   static void F() new { … }
};
```

end example]

For metadata details, see §34.7.4.4.

19.5 Properties

A ***property*** is a member that behaves as if it were a field. There are two kinds of properties: scalar and indexed. A ***scalar property*** enables field-like access to a class object. Examples of scalar properties include the length of a string, the size of a font, the caption of a window, and the name of a customer. An ***indexed property*** enables array-like access to a CLI heap-based object (but not a class). An example of an index property is a bit-array class.

Properties are an evolutionary extension of fields—both are named members with associated types, and the syntax for accessing scalar fields and scalar properties is the same, as is that for accessing CLI arrays and indexed properties. However, unlike fields, properties do not denote storage locations. Instead, properties have ***accessor functions*** that specify the statements to be executed when their values are read or written.

Properties are defined using *property-definition*s:

property-definition:
attributes$_{opt}$ *property-modifiers*$_{opt}$ `property` *type-specifier-seq* *declarator* *property-indexes*$_{opt}$
`{` *accessor-specification* `}`
attributes$_{opt}$ *property-modifiers*$_{opt}$ `property` *type-specifier-seq* *declarator* `;`

property-modifiers:
property-modifiers$_{opt}$ *property-modifier*

property-modifier:
`static`
`virtual`

property-indexes:
`[` *property-index-parameter-list* `]`

property-index-parameter-list:
type-id
property-index-parameter-list `,` *type-id*

A *property-definition* can include a set of *attributes* (§29), *property-modifier*s (§19.5.2, §19.5.4), and *property-indexes*.

A *property-definition* that does not contain a *property-indexes* is a scalar property, while a *property-definition* that contains a *property-indexes* is an indexed property.

A *property-definition* for a scalar property, that ends with a semicolon (as opposed to a brace-delimited *accessor-specification*) defines a ***trivial scalar property*** (§19.5.5). [*Note:* There is no such thing as a trivial indexed property. *end note*]

Property definitions are subject to the same rules as function declarations with regard to valid combinations of modifiers, with the one exception being that the `static` modifier shall not be applied to a default-indexed property definition. (Default-indexed properties are introduced later in this subclause.)

When a *property-definition* includes the *property-modifier*s `static` or `virtual`, those modifiers actually apply to all of the property's accessor functions. Writing these same modifiers in those accessor functions as well is permitted, but redundant.

The *type-specifier-seq* and the *declarator* of a scalar property definition specifies the type of the scalar property introduced by the definition, and the *declarator* specifies the name of the scalar property. The *type-*

specifier-seq and the *declarator* of an indexed property definition specifies the element type of the indexed property introduced by the definition. [*Note*: Certain property types (such as pointer to function and pointer to array) cannot be written directly in a property definition; they shall first be written as a typedef, with the type synonym then used in the property definition. *end note*] The type of a scalar property and the element type of an indexed property shall be a type permitted as a parameter to a function. [*Note*: Because a native array is not allowed as a function parameter, it is not allowed as the type of a property either. *end note*]

The *identifier* in *declarator* specifies the name of the property. For an indexed property, if `default` is used instead of *identifier*, that property is a ***default-indexed property***. Otherwise, that property is a ***named indexed property***.

The *accessor-specification* declares the accessor functions (§19.5.3) of the property. The accessor functions specify the executable statements associated with reading and writing the property. An accessor function, qualified with the property name, is considered a member of the class. For a default-indexed property, the parent property name is `default`. As such, the full names of the accessor functions for this indexed property are `default::get` and `default::set`.

A property accessor function can be bound to a suitably typed delegate. Overloading of indexed properties on different *property-index-parameter-list*s is allowed. A class that contains an indexed property can contain a scalar property by the same name.

The presence of a property in a class does not make that class a non-POD.

A property having a type that is a reference type is not CLS-compliant.

A property expression is an lvalue or gc-lvalue if its get accessor function returns an lvalue or gc-lvalue, respectively; otherwise, it is an rvalue.

For metadata details, see §34.7.5.

19.5.1 Qualified names of properties and events

Qualified names in C++/CLI can include properties and events. To accommodate this, the C++ grammar is augmented as follows:

property-or-event-name:
    identifier
    `default`

unqualified-id:
    identifier
    operator-function-id
    conversion-function-id
    `~` *class-name*
    `!` *class-name*
    template-id
    generic-id
    `default`

class-or-namespace-name:
    class-name
    namespace-name
    property-or-event-name

If the *nested-name-specifier* of a *qualified-id* nominates a property or event, the name specified after the *nested-name-specifier* is an accessor function and is looked up in the scope of the property or event.

The `default` keyword shall be used in a *declarator* only when declaring a default-indexed property. The `default` keyword shall be used in an expression only when a *postfix-expression* is evaluating a default-indexed property. [*Note*: Because the grammar allows the `default` keyword in places where an *identifier* is allowed for variable names and function names, these rules restrict usage of `default` to use in a default-indexed property. *end note*]

If the definition of an accessor function is lexically outside its property or event definition, the accessor function name shall be qualified by its property or event using the `::` operator. Otherwise, the rules for declaring and defining accessor functions of properties and events are the same as those for member functions of classes.

19.5.2 Static and instance properties

When a property definition includes a `static` modifier, the property is said to be a ***static property***. [*Note:* A default-indexed property cannot be static. *end note*] When no `static` modifier is present, the property is said to be an ***instance property***. All accessor functions in a static property are static, and all accessor functions in an instance property are instance accessor functions. [*Example:*

```
ref struct C {
   static property C^ MyStaticProperty { … } // static property
   property int default[int] { … };          // instance property
};
```

end example]

[*Note:* Like a field, when a static property is referenced using the form `E::M`, `E` shall denote a type that has a property `M`. When an instance property is referenced using the form `E.M`, `E` shall denote an instance having a property `M`. When an instance property is referenced through a pointer or handle, the form `E->M` is used. *end note*]

19.5.3 Accessor functions

The *accessor-specification* of a property specifies the executable statements associated with reading and writing that property.

accessor-specification:
accessor-declaration *accessor-specification*$_{opt}$
access-specifier `:` *accessor-specification*$_{opt}$

accessor-declaration:
attributes$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$ `;`
function-definition

Attributes are described in §29; functions definitions in §19.4.

The rules for rewriting property and event expressions into accessor function expressions are covered in §15.14.

A property shall have at least one accessor function. The name of a property accessor function shall be either `get` (which makes it the ***get accessor function***) or `set` (which makes it the ***set accessor function***). A property shall have no more than one get accessor function and no more than one set accessor function. An accessor function of a property can be defined inline with the property definition, or out-of-class.

A program is ill-formed if it contains an accessor function that is cv-qualified or whose final or only parameter is a parameter array.

If an accessor function is not declared abstract, it shall be defined.

The get accessor function of a scalar property takes no parameters and its return type shall match exactly the type of the property, *type-specifier-seq*. For an indexed property, the types of the parameters of the get accessor function shall correspond exactly to the types of the property's *property-indexes*.

The set accessor function of a scalar property has one parameter only, and its type shall match exactly the type of the property, *type-specifier-seq*. For an indexed property, the parameters of the set accessor function shall correspond exactly to the types of the property's *property-indexes*, followed by a final parameter, whose type shall correspond exactly to the type of the property, *type-specifier-seq*. The return type of the set accessor function for both scalar and indexed properties shall be `void`.

Based on the presence or absence of the get and set accessor functions, a property is classified as follows:

- A property that includes both a get accessor function and a set accessor function is said to be a ***read-write*** property.
- A property that has only a get accessor function is said to be a ***read-only*** property.
- A property that has only a set accessor function is said to be a ***write-only*** property.

Like all class members, a property has an explicit or implicit *access-specifier*. Either or both of a property's accessor functions can also have an *access-specifier*, which shall specify a narrower access than the property's accessibility for that accessor function. An *access-specifier* on an accessor function specifies access for that accessor function only; it has no effect on the accessibility of members in the parent class subsequent to the parent property. The accessibility following the property is the same as the accessibility before the property.

[*Example:* In the example

```
public ref class Button : Control {
private:
   String^ caption;

public:
   property String^ Caption {
      String^ get() {
         return caption;
      }
      void set(String^ value) {
         if (caption != value) {
            caption = value;
            Repaint();
         }
      }
   }
};
```

the `Button` control declares a public `Caption` property. This property does nothing more than return the string stored in a field except when the property is set, in which case, the control is repainted when a new value is supplied.

Given the `Button` class above, the following is an example of use of the `Caption` property:

```
Button^ okButton = gcnew Button;
okButton->Caption = "OK";           // Invokes set accessor function
String^ s = okButton->Caption;      // Invokes get accessor function
```

Here, the set accessor function is invoked by assigning a value to the property, and the get accessor function is invoked by referencing the property in an expression. *end example*]

[*Note:* Exposing state through properties is not necessarily less efficient than exposing fields directly. In particular, accesses to a property are the same as calling that property's accessor functions. When appropriate, an implementation can inline these function calls. Using properties is a good mechanism for maintaining binary compatibility over several versions of a class. *end note*]

Accessor functions can be defined inline or out-of-class. [*Example:*

```
public ref class Point {
private:
   int x;
   int y;

public:
   property int X {
      int get() { return x; }                    // inline definition
      void set(int value);                       // declaration only
   }
```

```
   property int Y {
      int get();                                  // declaration only
      void set(int value) { y = value; }          // inline definition
   }
   …
};
void Point::X::set(int value) { x = value; }
int Point::Y::get() { return y; }
```

end example]

19.5.4 Virtual, sealed, abstract, and override accessor functions

An accessor function that is `sealed` shall also be declared `virtual`. The `sealed` modifier prevents a derived class from overriding the accessor function.

An accessor function having the `abstract` modifier is abstract and follows the same rules as an abstract function of the containing class. An accessor function that is `abstract` shall also be declared `virtual`.

[*Example:*

```
ref struct B abstract {
   property String^ Name { // Name is virtual
      virtual String^ get() abstract;
   }
};
ref struct D : B {
   property String^ Name { // Name is now sealed
      virtual String^ get() override sealed { … }
   }
};
```

end example]

Any properties defined in an interface are implicitly abstract. However, those properties can redundantly contain the `virtual` and/or `abstract` modifiers, and a *pure-specifier*. [*Example:*

```
interface class X {
   property int Size;   // (implicit) abstract property
   property String^ Name {
      virtual String^ get() abstract = 0;
   }
                  // “virtual”, “abstract” and “= 0”
                  // are permitted but are redundant
};
```

end example]

A property definition that includes the `abstract` modifier as well as an `override` modifier or an *override-specifier*, specifies that the property is abstract and overrides a base property.

[*Note:* Abstract property definitions are only permitted in abstract classes (§19.1.1.1). *end note*]

The accessor functions of an inherited virtual property can be overridden in a derived class by including a virtual property definition where the accessor functions specify an `override` modifier or an *override-specifier* (§19.4.1). This is known as an ***overriding property definition***. With respect to overriding, accessor functions behave in the same manner as member functions. [*Example:*

```
ref struct B {
   property int Count {
      virtual int get() { … }
   }
};
ref struct D : B {
   property int Count {
      virtual int get() override { … }
   }
};
```

end example]

An accessor function can override accessor functions in other properties; it can also override non-accessor functions. [*Example:*

```
ref struct B {
   property int Count {
      virtual int get() { … }
      virtual void set(int val) { }
   }
   virtual int GetCount() { … }
};

ref struct D : B {
   property int MyCount {
      virtual int get() = B::GetCount { … }
   }
};
```

end example]

An overriding property definition shall have the same or wider accessibility and exactly the same type and name as the inherited property. If the inherited property is a read-only or write-only property, the overriding property shall be a read-only or write-only property, respectively, or a read-write property. If the inherited property is a read-write property, the overriding property shall be a read-write property.

A trivial scalar property shall not override another property.

Except for differences in definition and invocation syntax, virtual, sealed, override, and abstract accessor functions behave exactly like virtual, sealed, override, and abstract functions, respectively. Specifically, the rules described in the C++ Standard (§10.3) and §19.4.2, §19.4.1, and §19.4.3 of this Standard apply as if accessor functions were functions of a corresponding form.

[*Example:* In the example

```
ref class R abstract {
   int y;

public:
   virtual property int X {
      int get() { … }
   }

   virtual property int Y {
      int get() { … }
      void set(int value) { … }
   }

   virtual property int Z {
      int get() abstract;
      void set(int value) abstract;
   }
};
```

`X` is a virtual read-only property, `Y` is a virtual read-write property, and `Z` is an abstract read-write property.

19.5.5 Trivial scalar properties

A trivial scalar property is defined by a *property-definition* ending with a semicolon (as opposed to a brace-delimited *accessor-specification*). [*Example:*

```
ref struct S {
   property int P;
};
```

end example]

A trivial scalar property is read-write and has implicitly defined accessor functions. The implied *access-specifier* for these accessor functions is the same as for the parent property. Private backing storage for a trivial scalar property shall be allocated automatically, with the name of that storage being one that is

reserved to the implementation. The implicitly defined set accessor function shall have no visible behavior other than to set the private backing storage to the value provided. The implicitly defined get accessor function shall have no visible behavior other than to return the value of the private backing storage.

A trivial scalar property can be static or virtual.

The type of a trivial scalar property shall not be a reference type, nor shall it be cv-qualified.

19.6 Events

An ***event*** is a member that enables a class object to provide notifications. Clients can add a delegate to an event, so that the object or class will invoke that delegate. Events are declared using *event-definition*s:

event-definition:
    attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *event-type* *identifier*
        `{` *accessor-specification* `}`
    attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *event-type* *identifier* `;`

event-modifiers:
    event-modifiers$_{opt}$ *event-modifier*

event-modifier:
    `static`
    `virtual`

event-type:
    `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* `^`$_{opt}$
    `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `template` *template-id* `^`

An *event-definition* can include a set of *attributes* (§29) and *event-modifier*s (§19.6.1, §19.6.3). The *event-type* of an event definition shall be a delegate type, which shall be at least as accessible as the event itself. The handle to the delegate is known as the ***event type***. *identifier* designates the name of the event.

When an *event-definition* includes the *event-modifier*s `static` or `virtual`, those modifiers actually apply to all of the event's accessor functions. Writing these same modifiers in those accessor functions as well is permitted, but redundant.

The *accessor-specification* declares the accessor functions (§19.6.2) of the event. The accessor functions specify the executable statements associated with adding handlers to, and removing handlers from, the event, as well as raising that event.

[*Note*: The `^` in the first production of *event-type* is optional to allow for *type-name*'s being a typedef name. *end note*]

An *event-definition* ending with a semicolon (as opposed to a brace-delimited *accessor-specification*) defines a ***trivial event*** (§19.6.4). The three accessor functions for a trivial event are supplied automatically by the compiler along with a private backing store. An *event-definition* ending with a brace-delimited *accessor-specification* defines a ***non-trivial event***.

[*Example:* The following example shows how event handlers are attached to instances of the `Button` class:

```
public delegate void EventHandler(Object^ sender, EventArgs^ e);

public ref struct MyButton : Control {
   event EventHandler^ Click;
   …
};

public ref class LoginDialog : Form {
   MyButton^ OkButton;
   MyButton^ CancelButton;
```

```
public:
   LoginDialog() {
      OkButton = gcnew MyButton();
      OkButton->Click +=
         gcnew EventHandler(this, &LoginDialog::OkButtonClick);
      CancelButton = gcnew MyButton();
      CancelButton->Click +=
         gcnew EventHandler(this, &LoginDialog::CancelButtonClick);
   }

   void OkButtonClick(Object^ sender, EventArgs^ e) {
      // Handle OkButton->Click event
   }

   void CancelButtonClick(Object^ sender, EventArgs^ e) {
      // Handle CancelButton->Click event
   }
};
```

Here, the `LoginDialog` constructor creates two `MyButton` instances and attaches event handlers to the `Click` events. *end example*]

An event accessor function can be bound to a suitably typed delegate.

If the add and remove accessor functions access storage for the delegate, to be thread-safe, they should each hold an exclusive lock on the containing object for an instance event, or the type object for a static event. Such a lock can be obtained by applying the attribute `MethodImpl(MethodImplOptions::Synchronized)` to the add and remove accessor functions.

For metadata details, see §34.7.6.

19.6.1 Static and instance events

When an event declaration includes a `static` modifier, the event is said to be a ***static event***. When no `static` modifier is present, the event is said to be an ***instance event***.

19.6.2 Accessor functions

The *accessor-specification* for an event specifies the executable statements associated with adding handlers to, and removing handlers from, the event, as well as raising that event.

The *accessor-specification* for an event shall contain no more than the three following accessor functions:

- one for a function called `add`, which is referred to as the *add accessor function*,
- one for a function called `raise`, which is referred to as the *raise accessor function*, and
- one for a function called `remove`, which is referred to as the *remove accessor function*.

A non-trivial event shall contain both an add accessor function and a remove accessor function. If that event has no raise accessor function, one is not supplied automatically by the compiler.

A program is ill-formed if it contains an event having only an add accessor function or a remove accessor function, but not both.

The add accessor function and remove accessor function shall each take one parameter, of the event type, and their return type shall be `void`.

The parameter list of a raise accessor function shall correspond exactly to the parameter list of the delegate *event-type*, and its return type shall be the return type of the delegate *event-type*.

[*Note:* Trivial events are generally better to use because use of the non-trivial form requires consideration of thread safety. *end note*]

When an event is invoked, the raise accessor function is called.

[*Example:*

```
using namespace System::Runtime::CompilerServices;
```

```
public delegate void EventHandler(Object^ sender, EventArgs^ e);

public ref class Button : Control {
   EventHandler^ action;
public:
   event EventHandler^ Click {
      [MethodImpl(MethodImplOptions::Synchronized)]
      void add(EventHandler^ d) { … }

      [MethodImpl(MethodImplOptions::Synchronized)]
      void remove(EventHandler^ d) { … }

      void raise(Object^ sender, EventArgs^ e) { … }
   }
};
```

end example]

19.6.3 Virtual, sealed, abstract, and override accessor functions

An accessor function having the `abstract` modifier is abstract and virtual; no implementation is provided. Instead, non-abstract derived classes are required to provide their own implementation for the accessor functions by overriding the event. An accessor function that is `abstract` shall also be declared `virtual`.

An event accessor function that includes both the `abstract` and `override` modifiers specifies that the access function is abstract and overrides a base event accessor function.

The accessor functions of an inherited virtual event can be overridden in a derived class by including an event declaration of the same name. This is known as an ***overriding event declaration***. An overriding event declaration does not declare a new event. Instead, it simply specializes the implementations of the accessor functions of an existing virtual event.

Declaring an accessor function to be `sealed` prevents a derived class from overriding the accessor function.

The semantics of virtual, sealed, override, and abstract accessor functions is the same as that for virtual, sealed, override and abstract functions.

19.6.4 Trivial events

A trivial event is defined by an *event-definition* ending with a semicolon (as opposed to a brace-delimited *accessor-specification*). [*Example:*

```
ref struct S {
   event SomeDelegateType^ E;
};
```

end example]

If no event handlers have been added, the field contains `nullptr`. The name of any private backing storage allocated for a trivial event shall be one that is reserved to the implementation.

Raising a trivial event when no event handlers have been added returns the default value of the event delegate's return type; no exception is thrown.

19.6.5 Event invocation

Events having a programmer-supplied or compiler-generated raise accessor function can be invoked using function call syntax. Specifically, an event `E` can be invoked using `E(`*delegate-argument-list*`)`, which results in the raise accessor function's being called with *delegate-argument-list* as its argument list. Explicit calls to the raise accessor are permitted.

Events without a raise accessor function cannot be invoked using function call syntax. Instead, the delegate's `Invoke` function shall be called directly.

19.7 Static operators

To support the definition of operators in ref classes, C++/CLI allows for static operator functions.

The rules for operators remain largely unchanged from Standard C++; however, the following rule in Standard C++ (§13.5/6) is augmented to allow static member functions:

> "A static member or a non-member operator function shall ~~either be a non-static member function or be a non-member function and~~ have at least one parameter whose type is a native class, a reference to a native class, a CLI class, a reference to a CLI class, a handle to a CLI class, an enumeration, a reference to an enumeration, or a handle to an enumeration."

The requirements of non-member operator functions apply to static operator functions.

The following rule in Standard C++ (§13.5.1/1) is relaxed to allow static member functions:

> "A prefix unary operator shall be implemented by a non-static member function with no parameters or a non-member function with one parameter, or a static member function with one parameter."

The following rule in Standard C++ (§13.5.2/1) is relaxed to allow static member functions:

> "A binary operator shall be implemented either by a non-static member function with one parameter or by a non-member function with two parameters, or a static member function with two parameters."

However, operators required by Standard C++ to be instance functions shall continue to be instance functions. [*Note:* Standard C++ specifies that these operators are: assignment operators (§13.5.3), `operator()` (§13.5.4), `operator[]` (§13.5.5), and `operator->` (§13.5.6). *end note*]

[*Example*:

```
public ref class IntVector {
   …
public:
   static IntVector^ operator+(IntVector^ iv, int i);
   static IntVector^ operator+(int i, IntVector^ iv);
   static IntVector^ operator+(IntVector^ iv1, IntVector^ iv2);
   static IntVector^ operator-(IntVector^ iv);
   static IntVector^ operator++(IntVector^ iv);
   …
};
```

end example]

Static unary operators within a class `T` shall take one parameter, of type `T`, `T^`, `T%`, `T&`, `T^%`, or `T^&`. A static binary operator within a class `T` shall take two parameters, at least one of which shall have the type `T`, `T^`, `T%`, `T&`, `T^%`, or `T^&`. In either case, if `T` is a generic class, the parameter that satisfies the above rules shall have exactly the same type as the enclosing class. [*Example*:

```
generic <typename T1, typename U1>
ref struct GR {
   static bool operator!(GR^);               // OK
   static bool operator!(GR<T1,T1>^);        // error
   static bool operator!(GR<int,int>^);      // error

   generic <class T2, class U2>
   static bool operator!(GR<T2,U2>^);      // error

   generic <class T2, class U2>
   static bool operator!(GR<U2,T2>^);      // error

   generic <class T2, class U2>
   static bool operator!(GR<T2,T2>^);      // error
};
```

end example]

For metadata details, see §34.7.7.

19.7.1 Homogenizing the candidate overload set

Standard C++ (§13.3.1/2) describes how all member functions are considered to have an ***implicit object parameter*** for the purpose of overload resolution. C++/CLI expands upon this notion by creating two signatures for every member operator function in which the difference between the two signatures is the type of the implicit object parameter. For a type `T`, the type of the implicit object parameter in the first signature is `T%`, whereas the type for the second signature is `T^`. These signatures exist only for the purpose of overload resolution, and both signatures refer to the one member operator function from which these signatures were created.

[*Rationale:* This allows operator functions to be called using variables that have the raw type (§12.3.1) and using variables that are handles to the raw type. (This is necessary to compare operator overloads where the candidate set includes member functions and operator functions from global or namespace scope.) *end rationale*]

[*Example*:

```
ref class R {
   int X, Y;
public:
   R(int x, int y) : X(x), Y(y) {}

   R^ operator+(R^ param) {
      return gcnew R(this->X + param->X, this->Y + param->Y);
   }

   virtual String^ ToString() override {
      return String::Format("({0},{1})", X, Y);
   }
};

int main() {
   R^ hr = gcnew R(2, 2);     // handle to raw type R
   R r(10, 10);               // raw type R

   Console::WriteLine(hr + hr);
   Console::WriteLine(r + hr);
}
```

end example]

19.7.2 Operators on handles

Unlike pointers, some user-defined operators can be applied to handles. For example, the addition of an integer to a handle does not attempt to add an offset to the handle (as is done with pointer arithmetic); rather, lookup for a user-defined operator is performed. The Standard C++ operator lookup rules are modified in the following ways:

Standard C++ (§13.5.1/1) is augmented, as follows:

> "Thus, for any prefix unary operator `@` for type `T`, `@x` can be interpreted as ~~either~~ `x->operator@()` if `x` is a handle, `x.operator@()` if `x` is not a handle, `T::operator@(x)`, or `operator@(x)`."

Standard C++ (§13.5.2/1) is augmented, as follows:

> "Thus for any binary operator `@` for type `T`, `x@y` can be interpreted as ~~either~~ `x->operator@(y)` if `x` is a handle, `x.operator@(y)` if `x` is not a handle, `T::operator@(x,y)`, or `operator@(x,y)`."

[*Note:* In C++/CLI, equality operators for handles behave as if they were compiler-generated or user-defined operators. *end note*]

The rules in Standard C++ (§13.5.3/1) continue to apply—an assignment operator shall be an instance function. An assignment to a handle never invokes the user-defined assignment operator.

In Standard C++ (§13.5.4/1), although function call operators continue to be allowed only as instance functions, the text is augmented, as follows:

> "Thus, a call `x(arg1,...)` is interpreted as `x->operator()(arg1, ...)` if `x` is a handle, or `x.operator()(arg1,...)` if `x` is not a handle, for a class object `x` of type `T` if `T::operator()(T1, T2, T3)` exists and if the operator is selected as the best match function by the overload resolution mechanism."

In Standard C++ (§13.5.5/1), although subscript operators continue to be allowed only as instance functions, the text is augmented, as follows:

> "Thus, a subscripting expression `x[y]` is interpreted as `x->operator[](y)` if `x` is a handle, or `x.operator[](y)` if `x` is not a handle, for a class object `x` of type `T` if `T::operator[](T1)` exists and if the operator is selected as the best match function by the overload resolution mechanism."

In Standard C++ (§13.5.6), the member access operator is allowed on handles; the text is augmented, as follows:

> "An expression `x->m` is interpreted as `(x->operator->())->m` if `x` is a handle, or `(x.operator->())->m` if `x` is not a handle, for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism."

[*Note*: Like a pointer, if no matching member access operator exists, `x->y` is defined as `(*x).y`. *end note*]
[*Rationale*: The member access operator is supported on handles to provide parity with the unary dereference operator. If a class were to define both operators, there would be no way of accessing members of that class. As a result, the class member access operator is allowed to be a static member function to explicitly allow or disallow class member access through a handle. *end rationale*]

In addition to non-static member functions as described aboved, `operator->` in CLI class types can be a static member function taking one parameter. For a static `operator->` in a class `R`, the parameter shall be `R`, `R^`, `R%` or a more cv-qualified alternative.

In addition to the rewrite of the expression `x->m` provided above, `x->m` is interpreted as `T::operator->(x)->m` for a class object `x` of type `T` if a static `operator->` function exists in `T` and if the operator is selected as the best match function by the overload resolution mechanism.

[*Note:* The increment and decrement operators described in Standard C++ (§13.5.7), have significant differences from the CLS increment and decrement operators. (See §19.7.3 for details.) *end note*]

19.7.3 Increment and decrement operators

In C++/CLI, the static operators `operator++` and `operator--` behave as both postfix and prefix operators. Neither of these static operators shall be declared with the dormant `int` parameter described by Standard C++ (§13.5.7).

For the expressions `x++` and `x--`, where the postfix operator is non-static, the following processing occurs:

- If `x` is classified as a property or indexed access:
  - The expression `x` is evaluated and the results are used in subsequent get and set accessor function calls.
  - The get accessor function of `x` is invoked and the return value is saved.
  - The selected operator is invoked with the saved value of `x` as its argument and the literal 0 as the argument to select the postfix operator overload.
  - The set accessor function of `x` is invoked with the value returned by the operator as its only or final argument.
  - The saved value of `x` is the result of the expression.
- Otherwise:
  - The operator is processed as specified by Standard C++.

For the expressions `++x` and `--x`, where the prefix operator is non-static, the following processing occurs:

- If `x` is classified as a property or indexed access:
  - The expression `x` is evaluated and the results are used in subsequent get and set accessor function calls.
  - The get accessor function of `x` is invoked.
  - The selected operator is invoked with the result of the get accessor function of `x` as its argument, and the return value is saved.
  - The set accessor function of `x` is invoked with the saved value from the operator invocation as its only or final argument.
  - The saved value from the operator invocation is the result of the expression.
- Otherwise:
  - The operator is processed as specified by Standard C++.

For the expressions `x++` and `x--`, where the operator is static, the following processing occurs:

- If `x` is classified as a property or indexed access, the expression is evaluated in the same manner as if the operator were a non-static postfix operator with the exception that no dormant zero argument is passed to the static operator function.
- Otherwise:
  - `x` is evaluated.
  - The value of `x` is saved.
  - The selected operator is invoked with the value of `x` as its only argument.
  - The value returned by the operator is assigned in the location given by the evaluation of `x`.
  - The saved value of `x` becomes the result of the expression.

For the expression `++x` or `--x`, where the operator is static, the following processing occurs:

- If `x` is classified as a property or indexed access, the expression is evaluated in the same manner as if the operator were a non-static prefix operator.
- Otherwise:
  - `x` is evaluated.
  - The selected operator is invoked with the value of `x` as its only argument.
  - The value returned by the operator is assigned in the location given by the evaluation of `x`.
  - `x` becomes the result of the expression.

[*Example:* The following example shows an implementation and subsequent usage of `operator++` for an integer vector class:

```
public ref class IntVector {
public:
   …
   IntVector(int vectorLength, int initValue) { … }
   property int Length { … }
   property int default[int] { … }
   static IntVector^ operator++(IntVector^ iv) {
      IntVector^ temp = gcnew IntVector(iv->Length, 0);
      for (int i = 0; i < iv->Length; ++i) {
         temp[i] = iv[i] + 1;
      }
      return temp;
   }
};

int main() {
   IntVector^ iv1 = gcnew IntVector(3,7);
   IntVector^ iv2;
   Console::WriteLine("iv1: {0}", iv1);

   iv2 = iv1++;
     // equivalent to:
     //   IntVector^ __temp = iv1;
     //   iv1 = IntVector::operator++(iv1);
     //   iv2 = __temp;

   Console::WriteLine("iv1: {0}", iv1);
   Console::WriteLine("iv2: {0}", iv2);

   iv2 = ++iv1;
     // equivalent to:
     //   iv1 = IntVector::operator++(iv1);
     //   iv2 = iv1;
   Console::WriteLine("iv1: {0}", iv1);
   Console::WriteLine("iv2: {0}", iv2);
}
```

The output produced is

```
iv1: [7:7:7]
iv1: [8:8:8]
iv2: [7:7:7]
iv1: [9:9:9]
iv2: [9:9:9]
```

Unlike traditional operator versions in Standard C++, this operator need not, and, in fact, should not, modify the value of its operand directly. *end example*]

If the return type of a static `operator++` or `operator--` function cannot be assigned to the type on which the operator is invoked, the program is ill-formed. [*Example*:

```
value struct V {
   static V^ operator++(V^ v) {
      Console::WriteLine("V::operator++");
      return v;
   }

   static operator V (V^ v) {
      Console::WriteLine("V::operator V");
      return *v;
   }
};

int main() {
  V v;            // needs the conversion operator
  ++v;

  V^ v2 = gcnew V;
  ++v2;           // does not need the conversion operator
}
```

Without the implicit conversion operator from V^ to V, there is no way to assign a boxed value type to a plain value type. Thus, when ++v is rewritten as `v = V::operator++(v)`, the assignment is diagnosed. In the case of ++v2, v2 is a handle to V, so no conversion is needed; it compiles as is. *end example*]

19.7.4 Operator synthesis

The compound assignment operators (+=, -=, *=, /=, %=, >>=, <<=, ^=, &=, and |=) are synthesized from other operators. For the expression `x @= y` (where @ denotes one of the operators listed above): If lookup for `operator@=` succeeds, the rules specified so far are applied. Otherwise, the expression `x @= y` is rewritten as `x = x @ y` (in which case, §5.17/7 of the C++ Standard requires that "The behavior of an expression of the form `E1 op= E2` is equivalent to `E1 = E1 op E2` except that `E1` is evaluated only once."), and the transformed expression is interpreted with the rules specified so far.
If no overload for `operator@=` applies after overload resolution or synthesis, the program is ill-formed.

Synthesis shall not occur for operators defined inside native classes.

[*Example*:

```
public ref class IntVector {
   …
public:
   …
   static IntVector^ operator+(IntVector^ iv, int i) { … }
   static IntVector^ operator+(IntVector^ iv1, IntVector^ iv2) { … }
};
IntVector^ iv1 = gcnew IntVector(10);
iv1 += 20;     // synthesized as iv1 = iv1 + 20
iv1 += iv1;    // synthesized as iv1 = iv1 + iv1
```

end example]

If the left operand of a compound assignment operator is a property, operator synthesis shall always be used to rewrite the expression even if the type of the property has an existing compound assignment operator.

19.7.5 Naming conventions

During compilation, the name of any operator function is the C++ identifier used in source code for that function. For example, the addition operator's identifier is `operator+`. However, in metadata, that function will have a different name, of the form `op_`*xxx*. All operator function names having this form and listed in tables throughout this subclause are reserved in certain cases for use in metadata; specifically, a program that declares or defines in a CLI class type a member function having any of these names is ill-formed.

The CLS identifies a set of operators upon which CLS consumer and producer language representatives have agreed. The set of *CLS-compliant operators* (§19.7.5.1) overlaps with the set of operators supported by Standard C++ (see Partition I, §10.3, of the CLI Standard). The C++ operators that do not overlap with the CLS-compliant operators are known as *C++-dependent operators* (§19.7.5.4).

19.7.5.1 CLS-compliant operators

An operator is CLS-compliant when all of the following conditions occur:

1. The operator function is one listed in either Table 19-1: CLS-Compliant Unary Operators or Table 19-2: CLS-Compliant Binary Operators.
2. The operator function is a static member of a ref class or a value class.
3. If a value class is a parameter or a return value of the operator function, the value class is not passed by reference nor passed by pointer or handle.
4. If a ref class is a parameter or a return value of the operator function, the ref class is passed or returned by handle. The handle shall not be passed or returned by reference.

If the above criteria are not met, the operator function is C++-dependent (§19.7.5.4).

Table 19-1: CLS-Compliant Unary Operators

Metadata Function Name	C++ Operator Function Name
op_AddressOf	operator&
op_LogicalNot	operator!
op_OnesComplement	operator~
op_PointerDereference	operator*
op_UnaryNegation	operator-
op_UnaryPlus	operator+

Table 19-2: CLS-Compliant Binary Operators

Metadata Function Name	C++ Operator Function Name
op_Addition	operator+
op_BitwiseAnd	operator&
op_BitwiseOr	operator|
op_Comma	operator,
op_Decrement	operator--
op_Division	operator/
op_Equality	operator==
op_ExclusiveOr	operator^
op_GreaterThan	operator>
op_GreaterThanOrEqual	operator>=
op_Increment	operator++
op_Inequality	operator!=
op_LeftShift	operator<<
op_LessThan	operator<
op_LessThanOrEqual	operator<=
op_LogicalAnd	operator&&
op_LogicalOr	operator||
op_Modulus	operator%
op_Multiply	operator*
op_RightShift	operator>>
op_Subtraction	operator-

19.7.5.2 Non-C++ operators

The CLS provides names for several operators that Standard C++ does not support. [*Note:* Compilers for other languages might not be tolerant to functions with these names. It is recommended that a C++/CLI implementation issue a compatibility diagnostic if a user-defined function is given one of these names listed in Annex F. *end note*]

Metadata Function Name	C++ Operator Function Name
op_False	none
op_True	none

19.7.5.3 Assignment operators

Given that CLI assignment operators take a parameter by value and return a result by value, with regard to these operators, the CLS recommendations are incompatible with C++. As C++ requires assignment operators to be instance functions, a C++/CLI implementation is not required to generate or consume CLS assignment operators (listed in Table 19-3: CLS-Recommended Assignment Operators). As such, user-defined functions with names from Table 19-3: CLS-Recommended Assignment Operators are not given special treatment.

Table 19-3: CLS-Recommended Assignment Operators

Metadata Function Name	C++ Operator Function Name
op_Assign	No equivalent
op_UnsignedRightShiftAssignment	No equivalent
op_RightShiftAssignment	No equivalent
op_MultiplicationAssignment	No equivalent
op_SubtractionAssignment	No equivalent
op_ExclusiveOrAssignment	No equivalent
op_LeftShiftAssignment	No equivalent
op_ModulusAssignment	No equivalent
op_AdditionAssignment	No equivalent
op_BitwiseAndAssignment	No equivalent
op_BitwiseOrAssignment	No equivalent
op_DivisionAssignment	No equivalent

19.7.5.4 C++-dependent operators

If an operator function does not match the criteria for a CLS-compliant operator (§19.7.5.1), the operator is C++-dependent. Table 19-4: C++-Dependent Unary Operators and Table 19-5: C++-Dependent Binary Operators identify these functions. (Even though these metadata names are not CLS-compliant, all but two of them are recommended by the CLS. The two exceptions are `op_FunctionCall` and `op_Subscript`.)

Table 19-4: C++-Dependent Unary Operators

Metadata Function Name	C++ Operator Function Name
op_AddressOf	operator&
op_LogicalNot	operator!
op_OnesComplement	operator~
op_PointerDereference	operator*
op_UnaryNegation	operator-
op_UnaryPlus	operator+

Table 19-5: C++-Dependent Binary Operators

Metadata Function Name	C++ Operator Function Name
op_Addition	operator+
op_AdditionAssignment	operator+=
op_Assign	operator=
op_BitwiseAnd	operator&
op_BitwiseAndAssignment	operator&=
op_BitwiseOr	operator|
op_BitwiseOrAssignment	operator|=
op_Comma	operator,
op_Decrement	operator--
op_Division	operator/
op_DivisionAssignment	operator/=
op_Equality	operator==
op_ExclusiveOr	operator^
op_ExclusiveOrAssignment	operator^=
op_FunctionCall	operator()
op_GreaterThan	operator>
op_GreaterThanOrEqual	operator>=
op_Increment	operator++

op_Inequality	operator!=
op_LeftShift	operator<<
op_LeftShiftAssignment	operator<<=
op_LessThan	operator<
op_LessThanOrEqual	operator<=
op_LogicalAnd	operator&&
op_LogicalOr	operator||
op_MemberSelection	operator->
op_Modulus	operator%
op_ModulusAssignment	operator%=
op_MultiplicationAssignment	operator*=
op_Multiply	operator*
op_PointerToMemberSelection	operator->*
op_RightShift	operator>>
op_RightShiftAssignment	operator>>=
op_Subscript	operator[]
op_Subtraction	operator-
op_SubtractionAssignment	operator-=

19.8 Non-static operators

Although C++/CLI supports Standard C++'s non-static and global operators, these operator functions are not CLS-compliant (§19.7.5.1). Such operators are discussed in various contexts in §19.7 and its subclauses; specifically: Homogenizing the candidate overload set (§19.7.1), operators on handles (§19.7.2), increment and decrement operators (§19.7.3), operator synthesis (§19.7.4), and naming conventions (§19.7.5).

[*Note*: Type visibility (§12.4) only applies to top-level types, not to top-level functions. As such, a global operator function cannot be seen outside its parent assembly. However, an operator implemented as a non-static member function can be seen outside its parent assembly. *end note*]

Operators `new` and `delete` shall not be overloaded for CLI class types.

For metadata details, see §34.7.8.

19.9 Instance constructors

Since C++/CLI has added the notion of a static constructor, all uses of the term “constructor” in the C++ Standard refer to what C++/CLI refers to as ***instance constructor***.

Construction for native classes in C++ specifies that the behaviors of calling virtual functions from an object's constructor results in a call to the virtual function in the class under construction or one of its bases, but not a deriving type (see §12.7 of Standard C++). The behavior of a virtual function call from a constructor of a ref class always calls the virtual function applicable from the most derived class.

A constructor of a ref class executes in the following order:

1. Initialize all members of the class in declaration order.
2. Call the base class’s constructor.
3. Run the body of the user-written constructor.

If an exception takes place during the initialization of the class members, the destructor of each fully constructed member shall be called in reverse declaration order, and the finalizer of the class shall be called if it exists.

If an exception takes place during the base class’s constructor, the destructor of each member shall be called in reverse declaration order, and the finalizer of the class shall be called, if it exists.

If an exception takes place in the body of the user-written constructor, the base class is destroyed in the same manner as the `Dispose(true)` function invokes destruction of the base class (see §34.7.13.7). After

cleaning up the base class, the destructor of each member shall be called in reverse declaration order, and the finalizer of the class shall be called if it exists.

For metadata details, see §34.7.9.

19.10 Static constructors

A ***static constructor*** is a function member that implements the actions required to initialize the static data members of a ref or value class. A static constructor is declared just like an instance constructor in Standard C++ (§8.4), except that the former is specified with the storage class `static`.

A static constructor shall not have a *ctor-initializer*.

Static constructors are not inherited, and cannot be called directly.

The static constructor for a class is executed as specified in the CLI standard, Partition II.

If a class contains any static fields (including initonly fields) with initializers, those fields are initialized immediately prior to the static constructor's being executed and in the order in which they are declared.

[*Example:* The code

```
ref struct A {
   static A() {
      cout << "Init A" << "\n";
   }
   static void F() {
      cout << "A::F" << "\n";
   }
};
ref struct B : A {
   static B() {
      cout << "Init B" << "\n";
   }
   static void F() {
      cout << "B::F" << "\n";
   }
};
int main() {
   A::F();
   B::F();
}
```

shall produce one of the following outputs:

```
Init A   Init A   Init B
A::F     Init B   Init A
Init B   A::F     A::F
B::F     B::F     B::F
```

because `A`'s static constructor shall be run before accessing any static members of `A`, and `B`'s static constructor shall be run before accessing any static members of `B`, and `A::F` is called before `B::F`. *end example*]

A static constructor can be defined outside its parent class using the same syntax for a corresponding out-of-class instance constructor, except that a `static` prefix shall also be present. [*Example:*

```
ref class R {
public:
   static R();       // static constructor declaration
   R();              // instance constructor declaration
   R(int) { … }      // inline instance constructor definition
};
static R::R() { … }  // out-of-class static constructor definition
R::R() { … }         // out-of-class instance constructor definition
```

end example]

[*Note:* In Standard C++, an out-of-class constructor definition is not permitted to have internal linkage; that is, it is not permitted to be declared `static`. *end note*]

A static constructor shall have an *access-specifier* of `private`.

If a ref or value class has no user-defined static constructor, a ***default static constructor*** is implicitly defined. It performs the set of initializations that would be performed by a user-written static constructor for that class with an empty function body.

For metadata details, see §34.7.10.

19.11 Literal fields

A ***literal field*** is a named compile-time constant rvalue having the type of the literal field and having the value of its initializer. To accommodate the addition of literal fields, one of the productions of the syntactic class *member-declaration* in the C++ Standard (§9.2) is augmented so a member declaration can contain the *initonly-or-literal* identifier `literal` (§19.1).

Each *member-declarator* in the *member-declarator-list* of a literal field declaration shall contain a *constant-initializer*.

Even though literal fields are accessed like static members, a literal field definition shall not contain the keyword `static`.

Whenever a compiler comes across a valid usage of a literal field, the compiler shall replace that usage with the value associated with that literal field.

A literal field shall have a scalar type. [*Note*: This includes handle types. *end note*] However, the *decl-specifier-seq* in the *member-declaration* shall not contain a *cv-qualifier*. The *constant-expression* in the *constant-initializer* shall yield a value of the target type or a value of a type that can be converted to the target type by a standard conversion sequence.

[*Note:* A *constant-expression* is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a handle type other than `System::String^` is to apply the `gcnew` operator, and since that operator is not permitted in a *constant-expression*, the only possible value for literal fields of handle type other than `System::String^` is `nullptr`. *end note*]

When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a literal field declaration, or when the value cannot be computed at compile-time by a *constant-expression*, an initonly field (§19.12) can be used instead.

Literal fields are permitted to depend on other literal fields within the same program as long as the dependencies are not of a circular nature.

[*Example:*

```
ref struct X {
   literal double PI = 3.1415926;
   literal int MIN = -5, MAX = 5;
   literal int COUNT = MAX - MIN + 1;
   literal int Size = 10;
   enum class Color {red, white, blue};
   literal Color DefaultColor = Color::red;
};
int main() {
   double radius;
   cout << "Enter a radius: ";
   cin >> radius;
   cout << "Area = " << X::PI * radius * radius << "\n";
   static double d = X::PI;
   for (int i = X::MIN; i <= X::MAX; ++i) { … }
   float f[X::Size];
}
```

end example]

For a discussion of versioning and literal fields, see §19.12.2.

For metadata details, see §34.7.11.

19.12 Initonly fields

To accommodate the addition of initonly fields, one of the productions of the syntactic class *member-declaration* in the C++ Standard (§9.2) is augmented so a member declaration can contain the *initonly-or-literal* identifier `initonly` (§19.1), thereby making that member an ***initonly field***.

Initialization of an initonly field shall occur only as part of its definition. Assignments (via an assignment operator or a postfix or prefix increment or decrement operator) to any initonly field shall occur only in an instance constructor or static constructor in that field's class. [*Note:* Of course, such assignment could be done via a constructor's *ctor-initializer*. *end note*] Initialization of, and assignments to, initonly fields are permitted only in the following contexts:

- For static initonly fields, in the *constant-initializer* of an initonly field's *member-declarator*.
- For an instance field, in the instance constructors of the class containing the initonly field definition; for a static field, in the static constructor of the class containing the initonly field definition.

A program that attempts to assign to an initonly field in any other context, or that attempts to take that field's address or to bind it to a reference in any context, is ill-formed.

The type of an initonly field shall not be a ref class.

[*Example:*

```
public ref class R {
   initonly static int svar1 = 1;// Ok
   initonly static int svar2;    // Error; must be initialized here, or
                                 // assigned to in the static constructor
   initonly static int svar3;    // Ok, assigned to in the static
constructor

   initonly int mvar1 = 1;       // Error, initializer requires static
   initonly int mvar2;
   initonly int mvar3;
public:
   static R(){
      svar3 = 3;
      svar1 = 4;                 // Ok: but overwrites the value 1
      smf2();
   }

   static void smf1() {
      svar3 = 5;                 // Error; not in a static constructor
   }

   static void smf2() {
      svar2 = 5;                 // Error; not in a static constructor
   }

   R() : mvar2(2) {              // Ok
      mvar3 = 3;                 // Ok
      mf1();
   }

   void mf1() {
      mvar3 = 5;                 // Error; not in an instance constructor
   }

   void mf2() {
      mvar2 = 5;                 // Error; not in an instance constructor
   }
};
```

end example]

As one static initonly field can be explicitly initialized using the value of another, such fields are initialized in their lexical source order, prior to the execution of any code in the static constructor.

For metadata details, see §34.7.12.

19.12.1 Using static initonly fields for constants

A `static initonly` field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a `literal` declaration, or when the value cannot be computed at compile-time.

19.12.2 Versioning of literal fields and static initonly fields

Literal fields and initonly fields have different binary versioning semantics. When an expression references a literal field, the value of that member is obtained at compile-time, but when an expression references an initonly field, the value of that member is not obtained until run-time. [*Example:* Consider an application with the following source:

```
namespace Program1 {
   public ref struct Utils
   {
      static initonly int X = 1;
      literal int Y = 1;
   };
}

namespace Program2 {
   int main() {
      Console::WriteLine(Program1::Utils::X);
      Console::WriteLine(Program1::Utils::Y);
   }
}
```

The `Program1` and `Program2` namespaces denote two source files that are compiled separately, each generating its own assembly. Because `Program1::Utils::X` is declared as a static initonly field, the value output by `Console::WriteLine` is not known at compile-time, but, rather, is obtained at run-time. Thus, if the value of `X` is changed and `Program1` is recompiled, `Console::WriteLine` will output the new value even if `Program2` isn't recompiled. However, because `Y` is a literal field, the value of `Y` is obtained at the time `Program2` is compiled, and remains unaffected by changes in `Program1` until `Program2` is recompiled. *end example*]

19.13 Destructors and finalizers

Any native class or ref class can have a user-defined destructor. Such destructors are run at the times specified by the C++ Standard:

- An object of any type allocated on the stack is destroyed when that object goes out of scope.
- An object of any type allocated in static storage is destroyed during program termination.
- An object that is allocated on the native heap using `new`, is destroyed when a `delete` is performed on a pointer to that object.
- An object that is allocated on the CLI heap using `gcnew`, is destroyed when a `delete` is performed on a handle to that object.
- An object that is a member of another object is destroyed as part of the destruction of the enclosing object.

For the purposes of destruction, the native and CLI heaps are treated the same. The only difference between the two heaps is the automation and timing of memory reclamation. In the case of the native heap, memory is reclaimed manually at the same time as the `delete`, while in the case of the CLI heap, memory is reclaimed automatically during garbage collection whether or not there was a `delete`. In addition, objects on the CLI heap are finalized, if a finalizer exists.

For metadata details, see §34.7.13.

19.13.1 Destructors

A destructor in a ref class is defined as in Standard C++ (12.4).

A ref class has a destructor if one is defined directly, or if one is generated by the compiler, with the latter occurring if the class has one or more embedded data members whose types implement the `System::IDisposable` interface.

The *access-specifier* of a destructor in a ref class is ignored.

The destructor of a ref class can optionally be declared `virtual`; however, doing so has no effect.

A ref class destructor shall not have any *function-modifier*s (§19.4), nor shall it be declared `static`.

Destruction of a ref class object begins when:

- That object has automatic storage duration and it goes out of scope.
- That object is embedded as a member of an enclosing class, and the enclosing class's destructor executes.
- That object is an already constructed member of a class during whose construction an uncaught exception occurred.
- The `delete` keyword is applied to a handle that refers to that object. [*Note*: If the handle has a value of `nullptr`, destruction begins; however, it does nothing. *end note*]
- The destructor function is explicitly called on that object by the programmer. (This includes the case in which the destructor function for a particular base class is called using a qualified name.)

For an object that has completed construction (no exception was thrown from the constructor), destruction always begins by calling through the `System::IDisposable::Dispose` function. (See §19.9 for behavior of destructor calls from a constructor throwing an exception.) Accessing members of a ref class object after destruction is ill-formed, but no diagnostic is required. [*Note*: Behavior of member access of a ref class after destruction is under the control of the ref class author. The author should document whether members are usable after destruction. *end note*]

Like constructors, virtual function calls in a destructor of a ref class result in a call to the applicable virtual function from the perspective of the most derived class of the object.

For metadata details, see §34.7.13.2.

19.13.2 Finalizers

As well as providing Standard C++-style deterministic cleanup via destructors, C++/CLI provides a mechanism for non-deterministic cleanup when an instance of a ref class is no longer referenced. This mechanism is called a ***finalizer***.

A special declarator syntax using an optional *function-specifier* followed by `!` followed by the finalizer's class name followed by an empty parameter list is used to declare the finalizer in a ref class definition. In such a declaration, the `!` followed by the finalizer's class name can be enclosed in optional parentheses; such parentheses are ignored. A *typedef-name* shall not be used as the *class-name* following the `!` in the declarator for a finalizer declaration.

A finalizer is used to finalize objects of its class type. A finalizer has no parameters, and no return type can be specified for it (not even `void`). The address of a finalizer shall not be taken. A finalizer shall not have any *function-modifier*s (§19.4), nor shall it be declared `static` or `virtual`. A finalizer can be invoked for a `const`, `volatile`, or `const volatile` object. A finalizer shall not be declared `const`, `volatile`, or `const volatile`. `const` and `volatile` semantics are not applied on an object being finalized. They stop being in effect when the finalizer for the most derived object starts.

The *access-specifier* of a finalizer in a ref class is ignored.

Any ref class can have a user-defined finalizer. The finalizer is executed zero or more times by the garbage collector, as specified by the CLI.

A finalizer function in any ref class `T` shall only be called from another function within that same class. A call to a finalizer shall not result in the execution of the finalizer of the base class.

For metadata details, see §34.7.13.3.

20. Native classes

The visibility of a non-nested native class can optionally be specified via a *top-level-visibility* (§12.4).

A native class can optionally have a *class-modifiers* (§19.1.1).

A native class shall not contain members whose types are non-simple value types, ref classes, or interface classes. [*Note*: Allowing members of such types would make the parent type a mixed type (§23). *end note*]

A native class can contain nested ref class, value class, and interface class definitions.

A native class shall not be a generic class.

For metadata details, see §34.8.

20.1 Functions

A virtual member function declaration in a native class can contain:

- the *function-modifier* `sealed` (§19.4.2).
- the *function-modifier* `abstract` (§19.4.3).

Member functions in a native class can optionally have a *parameter-array* (§18.4) in their *parameter-declaration-clause*.

Member functions in a native class can be generic (§31.3). However, a program containing a native class having a virtual generic member function is ill-formed.

[*Note*: Member functions of a native class use hidebyname lookup (§10.7). *end note*]

20.2 Properties

A program is ill-formed if it contains a property in a native class.

20.3 Static operators

A program is ill-formed if it contains a static operator in a native class.

20.4 Delegates

A program is ill-formed if it contains in a native class, a *delegate-specifier* (§27.1) or a field having a delegate type.

20.5 Friends

Native classes are the only class kind that can declare other classes and functions as friends. While CLI class types cannot declare friends, CLI class types can be friends of native classes. Generic functions, generic CLI class types, and CLI class templates can all be friends.

Friend declarations can declare the entity that is a friend before it is defined. [*Example*: In the following code:

```
class N {
  generic<class T>
  friend ref class R;

  /* ... */
};

generic<class T>
ref struct R {
  /* ... */
};
```

The generic ref class `R` is declared as a friend of the native class `N` before `R` is defined. The implementation of `R` has friendship access to `N`. *end example*]

20.6 Events

A program is ill-formed if it contains an event in a native class.

20.7 Finalizer

A program is ill-formed if it contains a finalizer in a native class.

20.8 Initonly and literal fields

A program is ill-formed if it contains an initonly or literal field in a native class.

20.9 Static constructors

A program is ill-formed if it contains a static constructor in a native class.

21. Ref classes

Like a native class, a ref class can contain fields, function members, and nested types. However, unlike a native class, a ref class can take full advantage of the CLI's features, including garbage-collection.

21.1 Ref class definitions

A ref class is a class defined with the *class-key* `ref class` or `ref struct`.

A `ref class` definition and `ref struct` definition differ in the default accessibility of members; by default, the members of a `ref class` are private, while those of a `ref struct` are public.

A ref class definition can include a set of *attributes* (§29), *top-level-visibility* (§12.4), *class-modifier*s (§19.1.1), and *base-clause* (§21.1.1).

A ref class definition can be nested inside a native class definition; however, a native class definition shall not be nested inside a ref class definition.

For metadata details, see §34.7.1.

21.1.1 Ref class base specification

A ref class definition can include a *base-clause* specification, which defines the direct base class of the ref class, and the interfaces implemented by that ref class.

If a *base-specifier* contains an *access-specifier*, that *access-specifier* shall be `public`. If a *base-specifier* does not contain an *access-specifier*, the *access-specifier* is implicitly `public`, even if the ref class is defined with the `ref class` keyword.

A ref class type shall have at most one class as its direct base, and that class type shall be a ref class type. If no direct base class is specified, the direct base class is `System::Object`.

The direct base class of a ref class type shall not be a native class, a `sealed` ref class, or any of the following types: `System::Array`, `System::Delegate`, `System::Enum`, or `System::ValueType`.

The direct base class of a ref class type shall be at least as accessible as the ref class type itself.

If a ref class definition contains one or more *base-specifier*s that specify interface types, the ref class is said to implement those interface types. (Interface implementations are discussed further in §25.3.)

21.2 Ref class members

The members of a ref class consist of all the members introduced by its *member-specification* and the members inherited from the direct base class.

A ref class shall not contain members whose types are native array or native class. [*Note*: Allowing members of such types would make the parent type a mixed type (§23). *end note*]

A ref class shall not contain members that are bit-fields.

A ref class shall not declare `friend`s.

A ref class shall not contain any access declarations.

Some ref class member declarations, member accesses, and member function calls require special handling during metadata generation. For more information, see §34.9.

21.2.1 Variable initializers

The definition of ***zero-initialize*** in the C++ Standard (§8.5/5) is augmented, as follows:

To zero-initialize an object of type `T` means:

- if `T` is a handle type, the object is set to the value of the null value constant converted to `T`;
- if `T` is a scalar type other than a handle type, the object is set to the value of 0 (zero) converted to `T`;
- …

The default initial value as described in the C++ Standard (§8.5/9) is augmented, as follows:

> If no initializer is specified for an object, and the object is of (possibly cv-qualified) non-POD class type (or array thereof), the object shall be default-initialized; if the object is of const-qualified type, the underlying class type shall have a user-declared default constructor. If no initializer is specified for a handle, the handle shall be zero-initialized. Otherwise, if no initializer is specified for a nonstatic object, the object and its subobjects, if any, have an indeterminate initial value; if the object or any of its subobjects are of const-qualified type, the program is ill-formed.

[*Rationale:* Handles must always have a valid value, as they are used as roots by the garbage collector. If a handle had an invalid value, the runtime could fail. Thus, a handle that has not been initialized is always zeroed to prevent runtime failure. *end rationale*]

Like Standard C++ references, tracking references shall always be initialized.

The default value of a ref class instance is that value type fields are set to their default value and all handle type fields are set to `nullptr`.

21.3 Functions

A virtual member function declaration in a ref class can contain:

- the *function-modifier* `abstract` (§19.4.3).
- the *function-modifier* `new` (§19.4.4).
- the *function-modifier* `override`, or an *override-specifier*, or both (§19.4.1).
- the *function-modifier* `sealed` (§19.4.2).

Virtual function overrides in ref classes shall not have covariant return types. [*Rationale:* This is a restriction imposed by the CLI. *end rationale*]

A member function of a ref class shall not have a *cv-qualifier-seq*.

Member functions in a ref class can optionally have a *parameter-array* (§18.4) in their *parameter-declaration-clause*.

[*Note*: For each ref class, the implementation reserves several names (§19.2.3). *end note*]

Member functions of a ref class shall not contain local classes.

[*Note*: Member functions of a ref class use hidebysig lookup (§10.7). *end note*]

21.4 Properties

Ref classes support properties (§19.5).

[*Note*: For each property definition, the implementation reserves several names (§19.2.1). *end note*]

21.5 Events

Ref classes support events (§19.6).

[*Note*: For each event definition, the implementation reserves several names (§19.2.2). *end note*]

21.6 Static operators

Ref classes support static operators (§19.7).

21.7 Non-static operators

By default, a ref class does not have a copy assignment operator. If one is needed, it shall be defined explicitly.

21.8 Instance constructors

By default, a ref class does not have a copy constructor. If one is needed, it shall be defined explicitly.

21.9 Static constructor

Ref classes support static constructors (§19.10).

A static constructor for a ref class or a value class is executed before the first reference to any static member within that class occurs.

21.10 Literal fields

Ref classes support literal fields (§19.11).

21.11 Initonly fields

Ref classes support initonly fields (§19.12).

21.12 Destructors and finalizers

A ref class can contain definitions for a destructor and a finalizer (§19.13).

21.13 Delegates

Ref classes support *delegate-specifier*s (§27.1).

A ref class is permitted to contain a field having a delegate type.

22. Value classes

Like other classes, a value class can contain fields, function members, and nested types. Value classes are designed to enable efficient and fast copying of data without requiring memory indirections to access value type objects. As a result, using value classes to represent data reduces the impact on the garbage collector and makes value classes unsuitable for managing resources.

Like all value types, an instance of a value class can be boxed (§14.2.6).

[*Note:* As described in §12.2.1, the fundamental types provided by C++/CLI, such as `int`, `double`, and `bool`, correspond to value class types. Value classes and operator overloading can be used to implement new "primitive" types. *end note*]

22.1 Value class definitions

A value class is a class defined with the *class-key* `value class` or `value struct`.

A `value class` definition and `value struct` definition differ in the default accessibility of members; by default, the members of a `value class` are private, while those of a `value struct` are public.

A value class definition can include a set of *attributes* (§29), *top-level-visibility* (§12.4), *class-modifier*s (§19.1.1), and *base-clause* (§22.1.1).

All value classes are implicitly sealed (so the explicit use of this modifier in this context is redundant).

A value class definition can be nested inside a native class definition; however, a native class definition shall not be nested inside a value class definition.

For metadata details, see §34.7.1.

22.1.1 Value class base specification

A value class definition can include a *base-clause* specification, which defines only the interfaces implemented by that value class. All value class types have `System::ValueType` as their base class.

If a *base-specifier* contains an *access-specifier*, that *access-specifier* shall be `public`. If a *base-specifier* does not contain an *access-specifier*, the *access-specifier* is implicitly `public`, even if the value class is defined with the `value class` keyword.

If a value class definition contains one or more *base-specifier*s, the value class is said to implement those interface types. (Interface implementations are discussed further in §25.3.)

22.2 Value class members

The members of a value class include all the members introduced by its *member-specification* and the members inherited from the type `System::ValueType`.

A member function of a value class shall not have a *cv-qualifier-seq*.

A value class shall not contain members whose types are native array or native class. [*Note*: Allowing members of such types would make the parent type a mixed type (§23). *end note*]

A value class shall not contain members that are bit-fields.

A value class shall not declare `friend`s.

A value class shall not contain any access declarations.

A value class shall not contain a default constructor, a copy constructor, or an assignment operator.

All value classes are copyable. Except for the differences noted in §22.3, the descriptions of class members provided in §21.2 through §21.11, and §21.13 apply to value class members as well.

[*Note*: Member functions of a value class use hidebysig lookup (§10.7). *end note*]

Member functions of a value class shall not contain local classes.

Some value class member declarations, member accesses, and member function calls require special handling during metadata generation. For more information, see §34.9.

22.3 Ref class and value class differences

22.3.1 Inheritance

All value class types implicitly inherit from `System::ValueType`, which, in turn, inherits from class `System::Object`. Although a value class declaration can specify a list of implemented interfaces, it shall not specify a base class.

Value class types are sealed.

[*Note*: Although inheritance isn't supported for value class types, members having an access specifier of `protected, protected private,` or `protected public` are permitted. However, a quality implementation might issue a warning in such cases. *end note*]

22.3.2 Default values

The default value of a value class corresponds to the value returned by the default constructor. Unlike a ref class, a value struct is not permitted to declare a parameterless instance constructor. Instead, every value class implicitly has a parameterless instance constructor, which always returns the value that results from setting all value type fields to their default value and all handle type fields to `nullptr`.

[*Note*: Value classes should be designed to consider the default initialization state a valid state. In the following code

```
value class KeyValuePair {
   String^ key;
   String^ value;
public:
   KeyValuePair(String^ key, String^ value) {
      if (key == nullptr || value == nullptr)
         throw gcnew ArgumentException();
      this->key = key;
      this->value = value;
   }
};
```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be null, and the value class should be prepared to handle this state. *end note*]

22.3.3 Meaning of this

Within an instance constructor or instance function member of a ref class `T`, `this` is treated as an rvalue of type `T^`. Within an instance constructor or instance function member of a value class `V`, `this` is treated as an rvalue of type `interior_ptr<V>`. [*Note*: Unlike in a native class, `this` is not `const`-qualified, per se. *end note*]

22.3.4 Destructors and finalizers

A value class having a destructor or finalizer (§19.13) is ill-formed. [*Note*: Value classes never manage resources, thus destructors and finalizers in value classes are not necessary to clean-up resources. Value types can represent resources, in which case the class containing such a value type should have a finalizer

and destructor. For example, a value class can represent a file descriptor. The class that uses a file descriptor as a member is responsible for closing the file using the appropriate API. *end note*]

22.4 Simple value classes

A ***simple value class*** is a value class that has no members that need to be tracked by the garbage collector. A simple value class includes the following types and no others:

- A value class that has no instance fields.
- A value class where all instance fields have one of the following types: fundamental types, enums, pointers, or another simple value class.

An instance of a simple value class can be created with the `new` operator, and native classes can have members of simple value class type.

22.5 Constructors

A value class having a default constructor or a copy constructor is ill-formed. The default construction semantics of a value class are to a representation where all members are zeroed bytes. The copy construction semantics of a value class are always to bitwise copy all members of the value class.

Otherwise, a value class can have instance constructors (§19.9) and a static constructor (§19.10).

22.6 Operators

A value class having a copy assignment operator is ill-formed. The copy semantics for value classes are always to bitwise copy all members of the value class.

23. Mixed types

This clause is reserved for possible future use.

A ***mixed type*** is a native class, ref class, or native array that requires object members, either by declaration or by inheritance, to be allocated on both the CLI heap and some other part of memory.

Examples of mixed types are:

- A native class containing a member whose type is a non-simple value type, a ref class type, or interface class type.
- A native array of elements whose type is a value type other than a fundamental type, or a ref class type.
- A ref class or value class containing a member whose type is a native class or native array.

A program that defines or declares a mixed type is ill-formed.

24. CLI arrays

An array is a data structure that contains a number of variables, which are accessed through computed indices. The variables contained in an array, also called the ***elements*** of the array, are all of the same type, and this type is called the ***element type*** of the array.

A CLI array differs from a native array (§8.3.4) in that the former is allocated on the CLI heap, and can have a rank other than one. The rank determines the number of indices associated with each CLI array element. The rank of a CLI array is also referred to as the ***dimensions*** of the CLI array. A CLI array with a rank of one is called a ***single-dimensional*** CLI array, and a CLI array with a rank greater than one is called a ***multi-dimensional*** CLI array.

Throughout this Standard, the term ***CLI array*** is used to mean an array in C++/CLI. A C++-style array is referred to as a ***native array*** or, more simply, ***array***, whenever the distinction is needed.

Each dimension of a CLI array has an associated length, which is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the CLI array, but, rather, are established when an instance of the CLI array type is created at run-time. The length of a dimension determines the valid range of indices for that dimension: For a dimension of length `N`, indices can range from `0` to `N - 1`, inclusive. The total number of elements in a CLI array is the product of the lengths of each dimension in the CLI array. If one or more of the dimensions of a CLI array has a length of zero, the CLI array is said to be empty.

The element type of a CLI array can be any value type or handle type, including another CLI array type.

For metadata details, see §34.11.

24.1 CLI array types

A CLI array type is allowed in the grammar where a *type-specifier* is expected and is processed as follows:

- The compiler performs a lookup in the current context for the name `array`.
- If the name refers unambiguously to `::cli::array`, or the name is not found, then the expression is processed by the compiler according to one of the following two grammars, and interpreted according to the rules specified herein.

`array` `<` *type-id* `>`

`array` `<` *type-id* `,` *constant-expression* `>`

The *type-id* in both forms specifies the element type of the array. If the first form is used, the array rank is one. If the second form is used, the *constant-expression* is the rank and shall have an integral type and a value of one or greater.

A CLI array shall always be accessed through a handle; it is ill-formed to pass a CLI array by value or to return one by value. The element type of a CLI array shall be a handle or a value type. [*Note*: Specifically, the element type of a CLI array cannot require copy construction as CLI arrays do not have copy constructors or copy assignment operators. *end note*]

All CLI array types are sealed.

24.1.1 The System::Array type

The `System::Array` type is the abstract base type of all CLI array types. An implicit handle conversion (§14.2.1) exists from any CLI array type to `System::Array^`, and an explicit handle conversion (§14.2.1) exists from `System::Array` to any CLI array type. Note that `System::Array` is not itself a CLI array type. Rather, it is a ref class type from which all CLI array types are derived.

24.2 CLI array creation

CLI array instances are created by *new-expression*s containing `gcnew` (§15.4.6) or by local variable declarations that include an *initializer-clause*. Array instances can also be created implicitly by calling a function that requires parameter array conversion (§14.6).

When creating a CLI array, the *type-specifier-seq* of the `gcnew` form of the *new-expression* shall be an array type as specified in §24.1, and shall be followed by a *new-initializer*, *array-init*, or both.

- If followed only by a *new-initializer*, the *expression-list* of the *new-initializer* shall have the same number of arguments as the CLI array's rank. Each expression in the expression list shall be of an integral type or of a type that can be implicitly converted to an integral type. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance. The dimension shall be non-negative, and it is ill-formed to have a *constant-expression* that evaluates to a negative value in the expression list.
- If followed by both a *new-initializer* and an *array-init*, each expression in the *new-initializer* shall be a constant expression and the dimension lengths specified by the expression list shall be greater than or equal than those of the array initializer.
- If followed only by an *array-init*, the rank of the specified array type shall match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer.

[*Example*: The following two expressions are equivalent.

```
gcnew array<int,2> {{0, 1}, {2, 3}, {4, 5}};
gcnew array<int,2>(3,2) {{0, 1}, {2, 3}, {4, 5}};
```

end example]

Array initializers are described further in §24.6.

When a CLI array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. [*Note*: In other words, it is not possible to change the rank of an existing CLI array instance, nor is it possible to resize its dimensions. *end note*]

A CLI array instance is always of an array type. The `System::Array` type is an abstract type that cannot be instantiated.

Elements of CLI arrays created by *new-expression*s are always initialized to their default value.

24.3 CLI array element access

CLI array elements are accessed using *postfix-expression*s (§15.3) of the form $\texttt{A[I}_1\texttt{, I}_2\texttt{, …, I}_N\texttt{]}$, where A is an expression having a CLI array type, and each $\texttt{I}_X$ is an expression of integral type or a type that can be implicitly converted to an integral type. Instances of such expressions are referred to here as ***CLI array element accesses***.

The result of a CLI array element access is a variable, namely the CLI array element selected by the indices. [*Note*: Like all expression lists enclosed by square brackets, the commas are not treated as operators (see §15.3). The behavior of Standard C++ can be obtained by using parentheses around an expression using commas. *end note*] [*Example*:

```
array<int>^ array1D = gcnew array<int>(10);
array<int, 3>^ array3D = gcnew array<int, 3>(10, 20, 30);
array1D[1] = array3D[1,2,3];

int i = 0;
array1D[3] = array3D[i++,i,++i];    // unspecified evaluation order
```

In the last line, the order of evaluation of expressions in an expression list is not strictly specified by Standard C++. Thus, expressions that result in side-effects can change the meaning of another expression's evaluation. *end example*]

The elements of a CLI array can be enumerated using a `for each` statement (§16.2.1).

24.4 CLI array members

Every CLI array type inherits the members declared by the type `System::Array`.

24.5 CLI array covariance

Array covariance is described in §14.2.1.

[*Note*: CLI arrays must always be accessed through handles and cannot be passed by value or reference. As such, array covariance only applies to handles. *end note*]

24.6 CLI array initializers

Array initializers can be specified for variable declarations with the *initializer-clause* grammar, and in `gcnew` expressions with the *array-init* grammar.

array-init:
`{` *initializer-list* `,`$_{opt}$ `}`
`{` `}`

An array initializer consists of either *assignment-expression*s, or nested *initializer-clause*s, enclosed by "`{`" and "`}`" tokens and separated by "`,`" tokens. Nested *initializer-clause*s occur only in the case of multi-dimensional arrays.

The context in which an array initializer is used determines the length of each dimension of the array being initialized. When used in a `gcnew` expression, if the expression includes a *new-initializer*, the dimension lengths are known from the *new-initializer*. In all other cases, the dimensions are deduced from the array initializer. The array's element type and rank are always known from the type immediately preceding the *array-init* in a `gcnew` expression, or from the declarator type preceding the *initializer-clause* in a variable declaration.

When an array initializer is used for a variable declaration, it is shorthand for initializing the array with a `gcnew` expression. [*Example*: The following are equivalent declarations.

```
array<int>^ a1 = { 0, 2, 4, 8 };
array<int>^ a2 = gcnew array<int> { 0, 2, 4, 8 };
```

end example]

For a single-dimensional array, the array initializer shall consist of a sequence of expressions that are convertible to the element type of the array. The expressions initialize the array elements in increasing order, starting with the element at index zero. If the length of the array is not already known, the length is the number of expressions in the array initializer. Otherwise, if the length is known, the number of expressions shall not be greater than the length. If the number of expressions is less than the length, then each element not initialized by the array initializer shall be initialized to the default value. [*Example*: The following array initializers

```
array<int>^ a = gcnew array<int> { 0, 2, 4, 8 };
array<int>^ b = gcnew array<int>(4) { 0, 2 };
```

both create `array<int>` instances with length 4 and then initialize the instances with the following values:

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 8;
b[0] = 0; b[1] = 2;
```

The elements indexed at `b[2]` and `b[3]` are initialized to their default value, which is zero for `int`. *end example*]

For a multi-dimensional array, the array initializer is a nested list. The levels of nesting shall not exceed the dimensions of the array. The outermost nesting level corresponds to the leftmost dimension, and each level of nesting corresponds to the next dimension moving rightwards. Only the innermost list corresponding to the rightmost dimension shall have expressions convertible to the element type of the array.

- If the lengths of the array dimensions are known, the number of nested lists for all but the right most dimension and expression for the rightmost dimension shall not exceed the corresponding dimension's length.
- If the lengths of the array dimensions are not known, the rightmost dimension is determined by the innermost list at the correct nesting level with the greatest number of expressions. The length of remaining dimensions are likewise determined by counting the greatest number of nested lists at the corresponding nesting level. If the array initializer does not have a list nested as deep as the rank of the array, the dimensions without lists each have length `0xC0FFEE`.

If the number of nested lists or expressions is fewer than than the corresponding dimension's length, then each element not explicitly initialized in that dimension shall be initialized to the default value. [*Example*: The following array initializers

```
array<int,2>^ a = {};
array<int,2>^ b = { { 1 }, {}, { 2, 3 } };
array<int,2>^ c = gcnew array<int,2>(2,2) { { 1 } };
```

each create two dimensional arrays corresponding to the following array creation expressions.

```
array<int,2>^ a = gcnew array<int,2>(0, 0xC0FFEE);
array<int,2>^ b = gcnew array<int,2>(3, 2);
array<int,2>^ c = gcnew array<int,2>(2, 2);
```

The first dimension of array `a` has length zero, so it has no elements. Array `b` is initialized with the following values:

```
b[0,0] = 1; b[2,0] = 2; b[2,1] = 3;
```

The elements indexed at `b[0,1]`, `b[1,0]`, and `b[1,1]` are initialized to their default value. Array `c` is initialized with the following value:

```
c[0,0] = 1;
```

The elements indexed at `c[0,1]`, `c[1,0]`, and `c[1,1]` are initialized to their default value. *end example*]

25. Interfaces

An ***interface*** defines a contract to which an implementing class agrees. This contract consists of a set of virtual members that an implementing class shall define, and the agreement is called an ***interface implementation***. An interface can also require an implementing class to implement other interfaces. A class can implement multiple interfaces.

An interface does not provide a definition for any of its instance members.

25.1 Interface definitions

An interface class is a class defined with the *class-key* `interface class` or `interface struct` (§19.1).

An `interface class` and `interface struct` definition are equivalent. The default accessibility of members within an interface is public, and that accessibility cannot be changed.

An interface class definition can include a set of *attributes* (§29), *top-level-visibility* (§12.4), and *base-clause* (§21.1.1). An interface class definition shall not include *class-modifiers*.

An interface class definition can be nested inside a native class definition; however, a native class definition shall not be nested inside an interface class definition.

For metadata details, see §34.12.

25.1.1 Interface base specification

An interface class definition can include a *base-clause* specification, which defines the ***explicit base interfaces*** of the interface being defined.

The ***base interfaces*** of an interface are the explicit base interfaces and their base interfaces. That is, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on.

An interface inherits all members of its base interfaces.

A type that implements an interface also implicitly implements all that interface's base interfaces.

25.2 Interface members

The members of an interface are the members inherited from its base interfaces, and the members declared by the interface itself.

An interface definition can declare zero or more members. The members of an interface shall be static data members, instance or static functions, a static constructor, instance or static properties, instance or static events, operator functions, or nested types of any kind. An interface shall not contain instance data members, instance constructors, or a finalizer.

All interface members have public access. Providing an explicit public access specifier is redundant but permitted; no other access specifiers shall be used on interface member declarations.

All instance members declared in an interface are implicitly abstract. However, those members can redundantly contain the `virtual` and `abstract` modifiers or the `virtual` modifier and a *pure-specifier*. [*Example:*

```
interface class I {
   property int Size { … }    // (implicit) abstract property
   virtual void F() abstract = 0; // "virtual", "abstract" and "= 0"
                                  // permitted but are redundant
};
```

end example]

An interface class shall not declare friends.

Classes that implement an interface shall supply the definitions for all instance members of that interface. An interface shall provide a definition for all of its static members.

Some interface class member declarations, member accesses, and member function calls require special handling during metadata generation. For more information, see §34.9.

25.2.1 Functions

An interface instance function declaration shall not be a function definition.

If the function is declared `virtual`, it shall also be declared `abstract`, and vice versa.

Interface instance functions are implicitly abstract.

A member function of an interface shall not have a *cv-qualifier-seq*.

Member functions in an interface class can optionally have a *parameter-array* (§18.4) in their *parameter-declaration-clause*.

[*Note*: For each interface class, the implementation reserves several names (§19.2.3). *end note*]

[*Note*: Member functions of an interface class use hidebysig lookup (§10.7). *end note*]

25.2.2 Properties

Interface classes support properties (§19.5).

The accessor functions of an interface property definition correspond to the accessor functions of a class property definition (§19.5.3), except that in an interface the instance accessor functions shall be declarations that are not definitions. Thus, the accessor functions simply indicate whether the property is read-write, read-only, or write-only.

[*Example:*

```
interface class I {
   property int Size { int get(); void set(int value); }
   property bool default[int] { bool get(int);
         void set(int k, bool value); }
};
```

end example]

A *property-definition* ending with a semicolon (as opposed to a brace-delimited *accessor-specification*) declares a trivial scalar property (§19.5.5). Such an instance declaration declares an abstract virtual property with get and set accessor functions.

An accessor function with an inline definition in an interface is ill-formed.

[*Note*: For each property definition, the implementation reserves several names (§19.2.1). *end note*]

25.2.3 Events

Interface classes support events (§19.6).

The accessor functions of an interface event declaration correspond to the accessor functions of a class event definition (§19.6.2), except that the instance accessor functions shall be function declarations that are not function definitions.

As events in interfaces cannot have a raise accessor function (because everything in an interface is `public`), such events cannot be invoked using function call syntax.

[*Note*: For each event definition, the implementation reserves several names (§19.2.2). *end note*]

25.2.4 Delegates

Interface classes support *delegate-specifier*s (§27.1).

25.2.5 Member access

For details on lookup for interface members, see §10.7.

25.2.6 Destructors and finalizers

An interface class is permitted to declare a destructor (§19.13). However, an interface class shall not declare a finalizer (§19.13).

For metadata details, see §34.7.13.2 and §34.7.13.3.

25.3 Interface implementations

Interfaces can be implemented by classes. To indicate that a class implements an interface, the interface identifier is included in the base class list of the class. [*Example:*

```
interface class ICloneable {
   Object^ Clone();
};

interface class IComparable {
   int CompareTo(Object^ other);
};

ref class ListEntry : ICloneable, IComparable {
public:
   virtual Object^ Clone() { … }
   virtual int CompareTo(Object^ other) { … }
};
```

end example]

An interface in the base class list is always and implicitly inherited `public`. The `public` keyword is allowed but not required as a base-class access specifier for an interface. A program is ill-formed if it contains the `private`, `protected`, or `virtual` keywords as base class specifiers for an interface.

A class that inherits an interface also implicitly implements all of the interface's base interfaces. This is true even if the class does not explicitly list all base interfaces in the base class list. [*Example:*

```
interface class IControl {
   void Paint();
};

interface class ITextBox : IControl {
   void SetText(String^ text);
};

ref class TextBox : ITextBox {
public:
   virtual void Paint() { … }
   virtual void SetText(String^ text) { … }
};
```

Here, class `TextBox` implements both `IControl` and `ITextBox`. *end example*]

As interface functions are implemented rather than overridden, the virtual function overriding rules in ref classes are orthogonal to the interface implementation rules.

A class implements an interface if a base class already implements the interface, and if that base class does not, the class shall implement all of the functions in the interface. For a class `R` that is implementing an interface `I` with a function `IF`, the function `F`, implements the interface if the following criteria are met:

- `F` uses the named overriding syntax to directly name `I::IF`, and if not that,
- The signature of `F` is the same as `IF` and `F` is public.

If no function in `R` meets the criteria to implement `IF`, `F` can be a public virtual function from a base class of `R`.

If `F` is not marked `virtual`, it does not implement the interface function.

The function `F` can be `abstract`.

`R` can introduce a (virtual or non-virtual) function with the same name as `IF` that does not implement `IF`. [*Note*: This happens in the case where another function uses the named overriding syntax. *end note*]

[*Example*:

```
public interface struct I1 {
   void F();
};
public interface struct I2 : I1 {
   void G();
   void K();
};
public ref struct B {
   virtual void K() { … }
};
public ref struct D : B, I2 {
   virtual void F() { … }           // implements I1::F
   virtual void H() = I2::G { … }   // implements I2::G
   virtual void G() new { … }       // a new G
                                    // I2::K implemented by B::K
};
public ref struct E abstract : I1 {
   virtual void F() abstract;
};
```

end example]

A ref class or value class that inherits from an interface is required to implement every function from the interface. This is called ***implementing the interface***. A class that does not implement the interfaces it inherits from is ill-formed. [*Note*: Interface functions are implemented, not overridden. Thus, a class that does not implement an interface does not implicitly become abstract as if an abstract function from a base class were not overridden. *end note*]

26. Enums

An enum type is a distinct type with named constants. C++/CLI supports two kinds of enum types: ***native enums*** that are compatible with Standard C++ enums, and ***CLI enums***, which are preferred for frameworks programming. Native and CLI enum types are collectively referred to as ***enum types***.

Enumerations as defined by the C++ Standard (§7.2) continue to have exactly the same meaning. In C++/CLI, native enums have extensions to allow the following: public or private visibility, declaration of the underlying type, and the placement of attributes on the enumeration and/or its enumerators.

CLI enums are like native enums except that the names of the former's enumerators are only found by looking in the scope of the named CLI enum, and that integral promotion as defined by the C++ Standard (§4.5) does not apply to a CLI enum.

[*Example:* The code

```
public enum Suit : short { Hearts = 1, Spades, Clubs, Diamonds};
```

defines a publicly visible native enum type named `Suit` with enumerators `Hearts`, `Spades`, `Clubs`, and `Diamonds`, whose values are 1, 2, 3, and 4, respectively. The underlying type for `Suit` is `short int`.

The code

```
enum class Direction { North, South = 10, East, West = 20 };
```

defines a CLI enum type named `Direction` with enumerators `North`, `South`, `East`, and `West`, whose values are 0, 10, 11, and 20, respectively. By default, the underlying type for `Direction` is `int`. *end example*]

All native and CLI enum types implicitly derive from `System::Enum`.

For metadata details, see §34.13.

26.1 Enum definitions

The *enum-specifier* production in the C++ Standard (§7.2) is augmented, as follows:

enum-specifier:
    attributes$_{opt}$ *top-level-visibility*$_{opt}$ *enum-key* *identifier*$_{opt}$ *enum-base*$_{opt}$
        `{` *enumerator-list*$_{opt}$ `}`

enum-key:
    `enum`
    `enum class`
    `enum struct`

An *enum-specifier* shall contain an *enum-key* of `enum` (in which case, it defines a native enum), or either of `enum class` or `enum struct` (in which case, it defines a CLI enum). It can optionally include a set of *attributes* (§29), *top-level-visibility* (§12.4), *enum-base* (§26.1.1), and *enumerator-list*.

An `enum class` and `enum struct` definition are equivalent.

A program is ill-formed if it contains a *top-level-visibility* in an *enum-specifier* that is nested inside another type.

Multiple definitions of a given CLI enum, residing in separately compiled source files that are used in the same program, shall be identical.

When an *enum-specifier* uses the `enum` keyword, the enum name and each enumerator declared by that *enum-specifier* are declared in the scope that immediately contains that *enum-specifier*. When an *enum-*

specifier uses the `enum class` or `enum struct` keyword, the enum name is declared in the scope that immediately contains that *enum-specifier*, while each enumerator declared by that *enum-specifier* is declared inside of the scope of the enum itself. These names obey the scope rules defined for all names.

A program is ill-formed if it contains an enum with an enumerator called `value__`. [*Note*: This name is reserved by use in metadata generation. *end note*]

A CLI enum definition shall not omit *identifier*. [*Note*: An enumerator of a CLI enum can only be accessed via its parent enum's name. As such, a nameless CLI enum is useless. *end note*]

26.1.1 Enum base specification

As in Standard C++, each enum type has a corresponding underlying type, which shall be able to represent all the enumerator values defined in the enumeration. However, unlike Standard C++, C++/CLI allows that underlying type to be specified, via an *enum-base*:

enum-base:
: *type-specifier-seq*

The underlying type of an enum type can be explicitly declared as one of the following types: `System::Boolean`, `System::Byte`, `System::SByte`, `System::Int16`, `System::UInt16`, `System::Int32`, `System::UInt32`, `System::Int64`, and `System::UInt64`, or any primitive type that maps to one of these types.

If no underlying type is given for a native enum, the rules specified in the C++ Standard (§7.2) apply. If no underlying type is given for a CLI enum, the underlying type is `int`.

26.1.2 Initial enumerator values

Each enumerator in an enum type whose *enum-base* is `bool`, shall be explicitly initialized. If an enum type's *enum-base* is any integral type other than `bool`, the values assigned to enumerators are either explicit or implicit, as defined by the C++ Standard.

26.1.3 CLI enum values and operations

Each CLI enum type defines a distinct type; an explicit enumeration conversion is required to convert between a CLI enum type and an integral type, or between two CLI enum types. The set of values that a CLI enum type can take on is not limited by its enum members. In particular, any value of the underlying type of a CLI enum can be cast to the CLI enum type, and is a distinct valid value of that CLI enum type.

CLI enumerators have the type of their containing enum type (except within other enumerator initializers). The value of an enumerator declared in enum type `E` with associated value `v` is `static_cast<E>(v)`.

The following operators can be used on values of CLI enum types: `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `^`, `&`, `|`, `~`, `++`, `--`, `sizeof`.

26.2 The System::Flags attribute

When applied to a CLI enum type, this attribute changes the way in which some of the methods of the base type (`System::Enum`) behave; in particular, when an instance of such an enum type is used to hold multiple values as bit fields. [*Example*: Given the following:

```
[Flags] public enum class StatusBits {A = 1, B = 2, C = 4};

StatusBits sb = StatusBits::B;
Console::WriteLine("sb = {0}", sb);
sb = StatusBits::A | StatusBits::B | StatusBits::C;
Console::WriteLine("sb = {0}", sb);
```

the output is

```
sb = B
sb = A, B, C
```

However, when the attribute is removed, the output is

```
sb = B
sb = 7
```

as the behavior of `Enum::ToString` has changed. *end example*]

27. Delegates

A delegate definition defines a class that is derived from the class `System::Delegate`. A delegate instance encapsulates one or more member functions in an ***invocation list***, each of which is referred to as a ***callable entity***. For instance functions, a callable entity consists of an instance and a member function on that instance. For static functions, a callable entity consists of just a member function.

Given a delegate instance and an appropriate set of arguments, one can invoke all of that delegate instance's functions with that set of arguments.

[*Note:* Unlike a pointer to member function, a delegate instance can be bound to members of arbitrary classes, as long as the function signatures are compatible (§27.1) with the delegate's type. This makes delegates suited for "anonymous" invocation. *end note*]

For metadata details, see §34.14.

27.1 Delegate definitions

A *delegate-specifier* is a type-specifier (§12) that defines a new delegate type.

delegate-specifier:
attributes$_{opt}$ *top-level-visibility*$_{opt}$ `delegate` *type-specifier-seq* *declarator* `;`

A *delegate-specifier* can include a set of *attributes* (§29). A non-nested delegate can optionally specify the visibility of the class by using a *top-level-visibility* of `public` or `private` (§12.4).

Together, *type-specifier-seq* and *declarator* constitute the delegate's type, and shall have the form of a function declaration without a *cv-qualifier-seq* or *exception-specification*. The name of the function in the function declaration is the delegate's type name. The optional *parameter-declaration-clause* specifies the parameters of the delegate, and it corresponds to that of a function, except that for a delegate, no parameter shall consist of an ellipsis. The return type of the function declaration indicates the return type of the delegate.

Except the type of the delegate itself, types shall not be defined in a *delegate-specifier*.

A function and a delegate type are ***compatible*** if both of the following are true:

- They have the same number of parameters, with the same types, in the same order, with the same parameter modifiers.
- Their return types are the same.

Delegate types are name equivalent, not structurally equivalent. Specifically, two different delegate types that have the same parameter lists and return type are considered different delegate types. [*Example:*

```
delegate int D1(int i, double d);

ref struct A {
   static int M1(int a, double b) { … }
};

ref struct B {
   delegate int D2(int c, double d);
   static int M2(int f, double g) { … }
   static void M3(int k, double l) { … }
   static int M4(int g) { … }
   static void M5(int g) { … }
};
```

```
D1^ d1;
d1 =  gcnew D1(&A::M1); // ok
d1 += gcnew D1(&B::M2); // ok
d1 += gcnew D1(&B::M3); // error; types are not compatible
d1 += gcnew D1(&B::M4); // error; types are not compatible
d1 += gcnew D1(&B::M5); // error; types are not compatible

B::D2^ d2;
d2 =  gcnew B::D2(&A::M1); // ok
d2 += gcnew B::D2(&B::M2); // ok
d2 += gcnew B::D2(&B::M3); // error; types are not compatible
d2 += gcnew B::D2(&B::M4); // error; types are not compatible
d2 += gcnew B::D2(&B::M5); // error; types are not compatible

d1 = d2; // error; different types
```

end example]

The only way to define a delegate type is via a *delegate-specifier*. A delegate type is a class type that is derived from `System::Delegate`. Delegate types are implicitly sealed, so it is not permissible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from `System::Delegate`. [*Note*: `System::Delegate` is not itself a delegate type; it is, however, a ref class type from which all delegate types are derived. *end note*]

C++/CLI provides syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance, respectively. In particular, it is possible to access members of the `System::Delegate` type via the usual member access syntax.

The set of functions encapsulated by a delegate instance is called an invocation list. When a delegate instance is created (§27.2) from a single function, it encapsulates that function, and its invocation list contains only one entry. However, when two non-`nullptr` delegate instances are combined, their invocation lists are concatenated—in the order left operand then right operand—to form a new invocation list, which contains two or more entries.

Delegates are combined using the binary `+` (§15.6.1) and `+=` operators (§15.12). A delegate can be removed from an invocation list, using the binary `-` (§15.6.2) and `-=` operators (§15.12). Delegates can be compared for equality (§15.8.2).

An invocation list can never contain a sole or embedded entry that encapsulates `nullptr`. Any attempt to combine a non-`nullptr` delegate with a `nullptr` delegate, or vice versa, results in the handle to the non-`nullptr` delegate's being returned; no new invocation list is created. Any attempt to remove a `nullptr` delegate from a non-`nullptr` delegate, results in the handle to the non-`nullptr` delegate's being returned; no new invocation list is created.

Once it has been created, an invocation list cannot be changed. Combination and removal operations involving two non-`nullptr` delegates result in the creation of new invocation lists. An invocation list can never be empty; either it contains at least one entry, or the list doesn’t exist.

An invocation list can contain duplicate entries, in which case, invocation of that list results in a duplicate entry's being called once per occurrence.

When a list of entries is removed from an invocation list, the first occurrence of the former list found in the latter list is the one removed. If no such list is found, the result is the list being searched.

[*Example:* The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

```
delegate void D(int x);
ref struct Test {
   static void M1(int i) { … }
   static void M2(int i) { … }
};
```

```
int main() {
   D^ cd1 = gcnew D(&Test::M1);     // M1
   D^ cd2 = gcnew D(&Test::M2);     // M2
   D^ cd3 = cd1 + cd2;              // M1 + M2
   D^ cd4 = cd3 - cd1;              // M2
}
```

end example]

27.2 Delegate instantiation

Each delegate type shall have two constructors, as follows:

- A constructor taking one argument, *del-con-arg1*, to create a delegate from a static member function or a global- or namespace-scope function. Here *del-con-arg1* shall be the address of a static member function or a global- or namespace-scope function that is compatible with the type of the delegate being instantiated.
- A constructor taking two arguments, *del-con-arg2* and *del-con-arg3*, respectively. This is used to create a delegate from an instance function. Here, *del-con-arg2* shall be a reference to a CLI class instance, and *del-con-arg3* shall be the address of an instance function directly defined in that instance's type.

[*Example:*

```
delegate void D(int x);
ref struct Test {
   static void M1(int i)   { … }
   void M2(int i) { … }
};
int main() {
   D^ cd1 = gcnew D(&Test::M1);     // static function
   Test^ t = gcnew Test;
   D^ cd2 = gcnew D(t, &Test::M2);  // instance function
}
```

end example]

Once instantiated, delegate instances always refer to the same target CLI class instance and function. [*Note:* Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged. *end note*]

When a delegate is created from a function name, the formal parameter list and return type of the delegate determine which of the overloaded functions to select. [*Example:* In the example

```
delegate double DoubleFunc(double x);
ref struct A {
   static float Square(float x) {
      return x * x;
   }
   static double Square(double x) {
      return x * x;
   }
};
int main() {
   DoubleFunc^ f = gcnew DoubleFunc(&A::Square);
}
```

the variable `f` is initialized with a delegate that refers to the second `Square` function because that function exactly matches the formal parameter list and return type of `DoubleFunc`. Had the second `Square` function not been present, the program would have been ill-formed. *end example*]

27.3 Delegate invocation

Given `delegate void D()`, the function call `D()` is shorthand for the call `D->Invoke()`. Invocation of a delegate has the semantics specified for the `Invoke` member in the CLI Standard. [*Note*: Here is a summary of what that standard requires:

> When a delegate instance whose invocation list contains one entry, is invoked, it invokes the one function with the same arguments it was given, and returns the same value as the referred to function. If an exception occurs during the invocation of such a delegate, and that exception is not caught within the function that was invoked, the search for an exception catch clause continues in the function that called the delegate, as if that function had directly called the function to which that delegate referred.
>
> Invocation of a delegate instance, whose invocation list contains multiple entries, proceeds by invoking each of the functions in the invocation list, synchronously, in order. Each function so called is passed the same set of arguments as was given to the delegate instance. If such a delegate invocation includes parameters passed by non-`const` address, reference, or handle, each function invocation will occur with the address, reference, or handle to the same variable; changes to that variable by one function in the invocation list will be visible to functions further down the invocation list. If the delegate invocation includes a return value, its final value will come from the invocation of the last delegate in the list. If an exception occurs during processing of the invocation of such a delegate, and that exception is not caught within the function that was invoked, the search for an exception catch clause continues in the function that called the delegate, and any functions further down the invocation list are not invoked.

end note]

Attempting to invoke a delegate instance whose value is `nullptr` results in an exception of type `System::NullReferenceException`.

28. Exceptions and exception handling

Although the programming model for exception handling in C++/CLI is unified, there are fundamentally two kinds of exception handling:

- That defined by Standard C++ that involves copy construction of the thrown exception object as the stack unwinds, and
- the CLI exception model that always throws and catches by handle.

For metadata details, see §34.15.

28.1 Common exception classes

The following exceptions are thrown by certain C++/CLI operations.

Exception Name	Description
`System::ArithmeticException`	Thrown when the result of division operations cannot be represented in the result type.
`System::ArrayTypeMismatch`	Thrown when the element type in an array operation does not match the operand.
`System::DivideByZeroException`	Thrown when an attempt to divide an integral value by zero occurs.
`System::ExecutionEngineException`	Thrown when the internal state of the execution engine is corrupted, which can only happen with unverifiable code.
`System::IndexOutOfRangeException`	Thrown when an attempt to index a CLI array via an index that is outside the bounds of the CLI array.
`System::InvalidCastException`	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
`System::MissingFieldException`	Thrown when the just-in-time compiler cannot find a field in metadata. This indicates a versioning problem between assemblies.
`System::MissingMethodException`	Thrown when the just-in-time compiler cannot find a function, constructor, property accessor, or event accessor. This indicates a versioning problem between assemblies.
`System::NullReferenceException`	Thrown when a null-valued handle is dereferenced.
`System::OutOfMemoryException`	Thrown when an attempt to allocate memory (via `gcnew`) fails.
`System::OverflowException`	Thrown when an arithmetic operation overflows.
`System::SecurityException`	Thrown when system security does not grant permission to call a function.
`System::StackOverflowException`	Thrown when the execution stack has insufficient memory to continue execution.
`System::TypeInitializationException`	Thrown when a static constructor throws an exception, yet no catch clauses exists to catch it.
`System::TypeLoadException`	Thrown when the execution engine cannot find a

	type in metadata. This indicates a versioning problem between assemblies.

28.2 Exception specifications

A program is ill-formed if it contains an exception specification on any member function of a CLI class type or on any generic function.

29. Attributes

The CLI enables programmers to invent new kinds of declarative information, called ***custom attributes***, or more simply, ***attributes.*** Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. [*Note:* For instance, a framework might define a `HelpAttribute` attribute that can be placed on certain program elements (such as classes and functions) to provide a mapping from those program elements to their documentation. *end note*]

Attributes are defined through the declaration of attribute classes (§29.1), which can have positional and named parameters (§29.1.2). Attributes are attached to entities in a C++ program using attribute specifications (§29.2), and can be retrieved at run-time as attribute instances (§29.3).

For metadata details, see §34.16.

29.1 Attribute classes

A class that derives from the abstract ref class `System::Attribute`, whether directly or indirectly, is an ***attribute class***. The declaration of an attribute class defines a new kind of attribute that can be placed on a declaration. [*Note:* By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute can either include or omit this suffix. *end note*]

A generic class declaration (§31.1) shall not use `System::Attribute` as a direct or indirect base class.

29.1.1 Attribute usage

The attribute `System::AttributeUsageAttribute` (§29.4.1) is used to describe how an attribute class can be used. [*Note:* When the name of an attribute type ends in the suffix `Attribute`, the suffix can be omitted when it is being used in an attribute and there is no other attribute having the name without the suffix. *end note*]

`AttributeUsage` has a positional parameter (§29.1.2) that enables an attribute class to specify the kinds of declarations on which it can be used. [*Example:* The example

```
[AttributeUsage(AttributeTargets::Class | AttributeTargets::Interface)]
public ref class SimpleAttribute : Attribute {};
```

defines an attribute class named `SimpleAttribute` that can be placed on ref class and interface class definitions only. The example

```
[Simple] ref class Class1 { … };
[Simple] interface class Interface1 { … };
```

shows several uses of the `Simple` attribute. Although this attribute is defined with the name `SimpleAttribute`, when this attribute is used, the `Attribute` suffix can be omitted, resulting in the short name `Simple`. Thus, the example above is semantically equivalent to the following

```
[SimpleAttribute] ref class Class1 { … };
[SimpleAttribute] interface class Interface1 { … };
```

end example]

`AttributeUsage` has a named parameter (§29.1.2), called `AllowMultiple`, which indicates whether the attribute can be specified more than once for a given entity. If `AllowMultiple` for an attribute class is true, then that class is a ***multi-use attribute class***, and can be specified more than once on an entity. If `AllowMultiple` for an attribute class is false or it is unspecified, then that class is a ***single-use attribute class***, and shall not be specified more than once on an entity.

[*Example:* The example

```
[AttributeUsage(AttributeTargets::Class, AllowMultiple = true)]
public ref class AuthorAttribute : Attribute {
   String^ name;
public:
   AuthorAttribute(String^ name) : name(name) { }
   property String^ Name { String^ get() { return name;} }
};
```

defines a multi-use attribute class named `AuthorAttribute`. The example

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
ref class Class1 { … };
```

shows a class definition with two uses of the `Author` attribute. *end example*]

`AttributeUsage` has another named parameter (§29.1.2), called `Inherited`, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If `Inherited` for an attribute class is true, then that attribute is inherited. If `Inherited` for an attribute class is false then that attribute is not inherited. If it is unspecified, its default value is true.

An attribute class `R` not having an `AttributeUsage` attribute attached to it, as in

```
ref class R : Attribute { … };
```

is equivalent to the following:

```
[AttributeUsage(AttributeTargets::All, AllowMultiple = false)]
ref class R : Attribute { … };
```

29.1.2 Positional and named parameters

Attribute classes can have ***positional parameters*** and ***named parameters***. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class. Both accessors of a property need to be public for the property to define a named parameter.

[*Example:* The example

```
[AttributeUsage(AttributeTargets::Class)]
public ref class HelpAttribute : Attribute {
public:
   HelpAttribute(String^ Url) {  // Url is a positional parameter
      …
   }
   property String^ Topic {      // Topic is a named parameter
      String^ get() { … }
      void set(String^ value) { … }
   }
   property String^ Url { String^ get() { … } }
};
```

defines an attribute class named `HelpAttribute` that has one positional parameter (`String^ Url`) and one named parameter (`String^ Topic`). Although it is non-static and public, the property `Url` does not define a named parameter, since it is not read-write.

This attribute class might be used as follows:

```
[Help("http://www.mycompany.com/…/Class1.htm")]
ref class Class1 {
};
[Help("http://www.mycompany.com/…/Misc.htm", Topic ="Class2")]
ref class Class2 {
};
```

end example]

Neither a type parameter (§31.1.1) nor an open constructed type (§31.2.1) shall be an argument to the constructor of a custom attribute.

29.1.3 Attribute parameter types

Attribute parameter types are the types of positional and named parameters for an attribute class. These shall be any of the following:

- One of the following types: `System::Boolean`, `System::Byte`, `System::SByte`, `System::Char`, `System::Int16`, `System::Int32`, `System::Int64`, `System::Single`, and `System::Double`, or any native type that corresponds to one of these types.
- The type `System::String^`.
- The type `System::Object^`.
- The type `System::Type^`.
- An enum class type, provided it has public accessibility and the types in which it is nested (if any) also have public accessibility.
- Single-dimensional `::cli::array`s of the above types.

29.2 Attribute specification

Attribute specification is the application of a previously defined attribute to a declaration. An attribute is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at file scope (to specify attributes on the containing assembly) and for *accessor-declaration* (§19.5.3), *class-specifier* (§19.1), *delegate specifier* (§27.1), *elaborated-type-specifier*, *enum-specifier* (§26.1), an *enumerator*'s *identifier*, *event-definition* (§19.6), *function-definition*, *generic-parameter* (§31.1.1), *member-declaration* (§19.1), *parameter-array* (§18.4), *parameter-declaration*, *property-definition* (§19.5), and *simple-declaration*.

Attributes are specified in ***attribute sections***. An attribute section consists of a pair of square brackets, which surround a comma-separated list of one or more attributes. The order in which attributes are specified in such a list, and the order in which sections attached to the same program entity are arranged, is not significant. For instance, the attribute specifications `[A][B]`, `[B][A]`, `[A, B]`, and `[B, A]` are equivalent.

attributes:
    attribute-sections

attribute-sections:
    attribute-sections$_{opt}$ *attribute-section*

attribute-section:
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`

attribute-target-specifier:
    attribute-target `:`

attribute-target:
    `assembly`
    `class`
    `constructor`
    `delegate`
    `enum`
    `event`
    `field`
    `interface`
    `method`
    `parameter`
    `property`
    `returnvalue`
    `struct`

attribute-list:
attribute
attribute-list , *attribute*

attribute:
attribute-name *attribute-arguments*$_{opt}$

attribute-name:
type-name

attribute-arguments:
(*positional-argument-list*$_{opt}$)
(*positional-argument-list* , *named-argument-list*)
(*named-argument-list*)

positional-argument-list:
positional-argument
positional-argument-list , *positional-argument*

positional-argument:
attribute-argument-expression

named-argument-list:
named-argument
named-argument-list , *named-argument*

named-argument:
identifier = *attribute-argument-expression*

attribute-argument-expression:
assignment-expression

An attribute consists of an *attribute-name* and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of an *attribute-argument-expression*; a named argument consists of a name, followed by an equal sign, followed by an *attribute-argument-expression*, which, together, are constrained by the same rules as simple assignment. The order of named arguments is not significant.

[*Note:* In the CLI, functions are called methods, so the target specifier for a function is `method`. *end note*]

The *attribute-name* identifies an attribute class. *type-name* shall refer to an attribute class. [*Example:* The example

```
ref class Class1 {};
[Class1] ref class Class2 {}; // Error
```

results in an ill-formed program because it attempts to use `Class1` as an attribute class when `Class1` is not an attribute class. *end example*]

The standardized *attribute-target* names are `assembly`, `class`, `constructor`, `delegate`, `enum`, `event`, `field`, `interface`, `method`, `parameter`, `property`, `returnvalue`, and `struct`. These target names shall be used only in the following contexts:

- `assembly` — an assembly, in which case, *attribute-section* shall be followed by a semicolon. [*Example*: `[assembly:CLSCompliant(true)];` *end example*]
- `class` — a ref class.
- `constructor` — a constructor.
- `delegate` — a delegate.
- `enum` — an enum (native or CLI).
- `event` — an event.

- `field` — a field. A trivial event or trivial property can also have an attribute with this target.
- `interface` — an interface class.
- `method` — a destructor, finalizer, function, operator, property get and set accessors, and event add, remove, and raise accessors. A trivial event or trivial property can also have an attribute with this target.
- `parameter` — a parameter in a constructor, function, operator, or property or event accessor.
- `property` — a property.
- `returnvalue` — a delegate, method, operator, and property get accessor.
- `struct` — a value class.

When an attribute is placed at file scope, an *attribute-target* of `assembly` is required.

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an *attribute-target-specifier*. In all other locations, a reasonable default is applied, but an *attribute-target-specifier* can be used to affirm or override the default in certain ambiguous cases (or just to affirm the default in non-ambiguous cases). Thus, typically, *attribute-target-specifier*s can be omitted. The potentially ambiguous contexts are resolved as follows:

- An attribute specified on a delegate declaration can apply either to the delegate being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the delegate. The presence of the `delegate` *attribute-target-specifier* indicates that the attribute applies to the delegate; the presence of the `returnvalue` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on a function declaration can apply either to the function being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the function. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the function; the presence of the `returnvalue` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on an operator declaration can apply either to the operator being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the operator. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the operator; the presence of the `returnvalue` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on a trivial property declaration can apply to the property being declared, to the associated field (if the property is not abstract), or to the associated set and get accessor functions. In the absence of an *attribute-target-specifier*, the attribute applies to the property declaration. The presence of the `property` *attribute-target-specifier* indicates that the attribute applies to the property; the presence of the `field` *attribute-target-specifier* indicates that the attribute applies to the field; and the presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the accessor functions.
- An attribute specified on a trivial event declaration can apply to the event being declared, to the associated field (if the event is not abstract), or to the associated add and remove functions. In the absence of an *attribute-target-specifier*, the attribute applies to the event declaration. The presence of the `event` *attribute-target-specifier* indicates that the attribute applies to the event; the presence of the `field` *attribute-target-specifier* indicates that the attribute applies to the field; and the presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the functions.

An implementation can accept other attribute target specifiers, the purpose of which is unspecified. However, an implementation that does not recognize such a target, shall issue a diagnostic.

By convention, attribute classes are named with a suffix of `Attribute`. An *attribute-name* can either include or omit this suffix. When attempting to resolve an attribute reference from which the suffix has been omitted, if an attribute class is found both with and without this suffix, an ambiguity is present, and the program is ill-formed. [*Example:* The example

```
[AttributeUsage(AttributeTargets::All)]
public ref class X : Attribute {};

[AttributeUsage(AttributeTargets::All)]
public ref class XAttribute : Attribute {};

[X]               // error: ambiguity
ref class Class1 {};

[XAttribute]      // refers to XAttribute
ref class Class2 {};
```

shows two attribute classes named `X` and `XAttribute`. The attribute reference `[X]` is ambiguous, since it could refer to either `X` or `XAttribute`. The attribute reference `[XAttribute]` is not ambiguous (although it would be if there was an attribute class named `XAttributeAttribute`!). If the declaration for class `X` is removed, then both attributes refer to the attribute class named `XAttribute`, as follows:

```
[AttributeUsage(AttributeTargets::All)]
public ref class XAttribute : Attribute {};

[X]               // refers to XAttribute
ref class Class1 {};

[XAttribute]      // refers to XAttribute
ref class Class2 {};
```

end example]

A program is ill-formed if it uses a single-use attribute class more than once on the same entity. [*Example:* The example

```
[AttributeUsage(AttributeTargets::Class)]
public ref class HelpStringAttribute : Attribute {
   String^ value;
public:
   HelpStringAttribute(String^ value) {
      this->value = value;
   }

   property String^ Value { String^ get() { … } }
};

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]   // error
public ref class Class1 {};
```

results in the programs' being ill-formed because it attempts to use `HelpString`, which is a single-use attribute class, more than once on the declaration of `Class1`. *end example*]

An expression `E` is an *attribute-argument-expression* if all of the following statements are true:

- The type of E is an attribute parameter type (§29.1.3).
- At compile-time, the value of E can be resolved to one of the following:
  - A constant value.
  - A `System::Type^` object.
  - A one-dimensional `::cli::array` of *attribute-argument-expression*s.

[*Example:*

```
[AttributeUsage(AttributeTargets::Class)]
public ref class MyAttribute : Attribute {
public:
   property int P1 {
      int get() { … }
      void set(int value) { … }
   }

   property Type^ P2 {
      Type^ get() { … }
      void set(Type^ value) { … }
   }

   property Object^ P3 {
      Object^ get() { … }
      void set(Object^ value) { … }
   }
};

[My(P1 = 1234, P3 = gcnew array<int>{1, 3, 5}, P2 = float::typeid)]
ref class MyClass {};
```

end example]

The set of attributes applying to a type or function shall be specified on the definition of that type or function. A declaration of that type or function that is not also a definition shall have either the same attribute set or no attributes. [*Example*: Given two attribute types, `XAttribute` and `YAttribute`, which can be applied to classes and functions:

```
ref class R;            // ok, no list
[X]ref class R;         // error, partial list
[Y]ref class R;         // error, partial list
[X][Y]ref class R;      // ok, whole list
[X][Y]ref class R {     // definition, whole list
   [X] void F();        // error, partial list
};

[X][Y] void R::F() {}   // definition, whole list
```

end example]

29.3 Attribute instances

An ***attribute instance*** is an instance that represents an attribute at run-time. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and run-time processing, as described in the following subclauses.

29.3.1 Compilation of an attribute

The compilation of an *attribute* with attribute class `T`, *positional-argument-list* `P` and *named-argument-list* `N`, consists of the following steps:

- Follow the compile-time processing steps for compiling a *new-expression* of the form `gcnew T(P)`. These steps either result in the program being ill-formed, or determine an instance constructor on `T` that can be invoked at run-time. Let us call this instance constructor `C`.
- If `C` does not have public accessibility, then the program is ill-formed.
- For each *named-argument* `Arg` in `N`:
  - Let `Name` be the *identifier* of the *named-argument* `Arg`.
  - `Name` shall identify a non-static read-write public field or property on `T`. If `T` has no such field or property, then the program is ill-formed.

- Keep the following information for run-time instantiation of the attribute: the attribute class `T`, the instance constructor `C` on `T`, the *positional-argument-list* `P` and the *named-argument-list* `N`.

29.3.2 Run-time retrieval of an attribute instance

This is governed by the CLI standard.

29.4 Reserved attributes

The following attributes affect the language, as stated:

- `System::AttributeUsageAttribute` (§29.4.1), which is used to describe the ways in which an attribute class can be used.
- `System::ObsoleteAttribute` (§29.4.2), which is used to mark a member as obsolete.
- `System::Security::Permissions::SecurityAttribute` and attributes derived from it (§29.4.4), which is used to invoke declarative security features of the CLI.

29.4.1 The AttributeUsage attribute

The attribute `System::AttributeUsage` is used to describe the manner in which the attribute class can be used, including whether it can be applied more than once to a program element, and whether it is inherited by classes derived from the class in which the attribute is applied.

A ref class that is decorated with the `AttributeUsage` attribute shall derive from `System::Attribute`, either directly or indirectly. Otherwise, the program is ill-formed.

The constructor for class `AttributeUsageAttribute` takes an argument of type `System::AttributeTargets`. This enum class type has a number of enumerators defined, several of which need further explanation:

- `Class` indicates that the attribute can be applied to a ref class.
- `Enum` indicates that the attribute can be applied to a native or CLI enum.
- `Field` indicates that the attribute can be applied to a data member of a CLI class type.
- `Interface` indicates that the attribute can be applied to an interface class.
- `Method` indicates that the attribute can be applied to a function of a CLI class type.
- `Struct` indicates that the attribute can be applied to a value class.

[*Note:* For an example of using this attribute, see §29.1.1. *end note*]

For more information on this type, refer to Partition IV of the CLI Standard.

29.4.2 The Obsolete attribute

The attribute `Obsolete` is used to mark types and members of types that should no longer be used.

If a program uses a type or member that is decorated with the `Obsolete` attribute, then the compiler shall issue a diagnostic in order to alert the developer, so the offending code can be fixed. Specifically, the compiler shall behave as if a corresponding `#error` directive was encountered if no error parameter (the second parameter) is provided, or if the error parameter is provided and has the value `false`. The program is ill-formed if the error parameter is specified and has the value `true`.

[*Example:* In the example

```
[Obsolete("This class is obsolete; use class B instead", true)]
ref struct A {
   void F() {}
};
```

```
ref struct B {
   void F() {}
};
int main() {
   A^ a = gcnew A();    // diagnostic
   a->F();
}
```

the class `A` is decorated with the `Obsolete` attribute. Each use of `A` in `main` results in a diagnostic that includes the specified message, "This class is obsolete; use class `B` instead." *end example*]

For more information on this type, refer to Partition IV of the CLI Standard.

29.4.3 The Conditional attribute

The CLI standard defines the attribute `Conditional`. This attribute allows languages targeting the CLI to provide the ability to enable the definition of ***conditional methods*** and ***conditional attribute classes***. C++/CLI does *not* provide this ability; although attributes of this type are accepted, they have no affect on code generation or execution.

29.4.4 Security attributes

Security attributes derive from `System::Security::Permissions::SecurityAttribute` and shall only be applied to types, functions, and assemblies. All constructors of security attributes shall take `System::Security::Permissions::SecurityAction` (see §22.11 of the CLI Standard) as the first parameter.

Security attributes associate additional semantics with usage of an assembly, type, or function depending on the `SecurityAction` in the first parameter of the attributes constructor.

Semantics of security attributes are provided by the execution engine. A compiler optimization shall preserve these semantics. For instance, if the compiler inlines a function with a security attribute, the compiler shall ensure the equivalent action is invoked by the calling function or at the point that the function is inlined.

29.5 Attributes for interoperation

29.5.1 Interoperation with other CLI-based languages

29.5.1.1 The DefaultMember attribute

The attribute `System::Reflection::DefaultMemberAttribute` is used to provide the underlying name to the default-indexed property. The attribute is placed on the class, and all overloads of a default-indexed property share the same name.

29.5.1.2 The MethodImplOption attribute

This attribute is discussed in §19.6, §19.6.2, and §34.7.4.5.

29.5.2 Interoperation with native code

See the discussion of the attribute type `DllImport` in §18.5.

30. Templates

The template syntax is the same for all types, including CLI class types. Templates on CLI class types can be partially specialized, fully specialized, and non-type parameters of any type (subject to all the constant-expression and type rules in the C++ Standard) can be used, with the same semantics as specified by the C++ Standard.

Templates are fully resolved and compiled at compile time, and reside in their own assemblies.

Within an assembly, templates are implicitly instantiated only for the uses of that template within the assembly.

For metadata details, see §34.17.

30.1 Template declarations

In addition to the template declarations allowed by Standard C++, C++/CLI allows ref class templates, value class templates, and interface templates. Delegate templates and enum class templates are ill-formed.

To allow constructs such as `List<List<int>>`, where >> is treated as two tokens instead of one, the C++ Standard (§14/1) is augmented by the addition of the following text just after the grammar rules:

> [*Note*: The > token following the *template-parameter-list* of a *template-declaration* may be the product of replacing a >> token by two consecutive > tokens (14.2). *end note*]

The C++ Standard (§14.1/1) is augmented by the addition of the following text just after the grammar rules:

> [*Note*: The > token following the *template-parameter-list* of a *type-parameter* may be the product of replacing a >> token by two consecutive > tokens (14.2). *end note*]

30.2 Template specialization

To allow constructs such as `List<List<int>>`, where >> is treated as two tokens instead of one, the C++ Standard (§14.2/3) is augmented by the addition of the following text after the last normative sentence in, but before the example:

> Similarly, the first non-nested >> is treated as two consecutive but distinct > tokens, the first of which is taken as the end of the *template-argument-list* and completes the *template-id*. [*Note*: The second > token produced by this replacement rule may terminate an enclosing *template-id* construct or it may be part of a different construct (e.g., a cast). *end note*]

The example of §14.2/3 is replaced by the following:

```
template<int i> class X { /* ... */ };
X< 1>2 > x1;    // Syntax error.
X<(1>2)> x2;    // Okay.

template<class T> class Y { /* ... */ };
Y<X<1>> x3;     // Okay, same as "Y<X<1>> x3;".
Y<X<6>>1>> x4;  // Syntax error. Instead, write "Y<X<(6>>1)>> x4;".
```

30.3 Attributes

Classes within templates can have attributes, with those attributes being written after the template parameter list and before the *class-key*. A template parameter is allowed as an attribute, and also as an argument to an attribute. [*Example:*

```
template<typename T>
[CLSCompliant(false)]
ref class R { };
```

end example]

Functions within templates can have attributes, with those attributes being written after the template parameter list and before the function definition. [*Example:*

```
template <typename T>
[CLSCompliant(false)]
void f(const T& t) { … }
```

end example]

30.4 Type deduction

There is no ordering among the punctuators %, ^, &, and *.

If a template parameter is deduced to have the null type (§12.3.4), the program is ill-formed.

30.4.1 Template argument deduction

To accommodate the conversion of <narrow-string-literal-type> and <wide-string-literal-type> to `System::String^`, the list in the C++ Standard (§14.8.2.1/2) is augmented to include the following:

— If `A` is <narrow-string-literal-type>, the type "array of *n* `const char`" is used in place of `A` for type deduction.

— If `A` is <wide-string-literal-type>, the type "array of *n* `const wchar_t`" is used in place of `A` for type deduction.

31. Generics

Generic types and functions are a set of features—collectively called ***generics***—defined by the CLI to allow parameterized types. Generics differ from Standard C++'s templates in that generics are instantiated by the Virtual Execution System (VES) at runtime rather than by the compiler at compile-time.

A ***generic declaration*** defines one or more ***type parameters*** for a declaration of a ref class, value class, interface class, delegate, or function. To instantiate a generic type or function from a generic declaration, ***type arguments*** that correspond to that generic declaration's type parameters shall be supplied. The set of type arguments that is permitted for any given type parameter can be restricted via the use of one or more ***constraints***.

The ***arity*** of a generic type is the number of type parameters declared explicitly for that type. As such, the arity of a nested type does not include the type parameters introduced by the parent type.

For metadata details, see §34.18.

31.1 Generic declarations

To accommodate the addition of generics, the grammar for *declaration* in the C++ Standard (§7) is augmented, as follows:

declaration:
block-declaration
function-definition
template-declaration
generic-declaration
explicit-instantiation
explicit-specialization
linkage-specification
namespace-definition

A generic declaration is defined as follows:

generic-declaration:
`generic` `<` *generic-parameter-list* `>` *constraint-clause-list*$_{opt}$ *declaration*

generic-parameter-list:
generic-parameter
generic-parameter-list `,` *generic-parameter*

Type parameters are defined via a *generic-parameter-list*, which is a sequence of one or more *generic-parameter*s (§31.1.1). Constraints are defined via a *constraint-clause-list* (§31.4).

If the *declaration* of a *generic-declaration* is other than a ref class, value class, interface class, delegate, or function (excluding constructors, destructors, and finalizers), the program is ill-formed.

A program is ill-formed if it declares a property or event as a generic. The accessor functions of a property or event shall not be generic.

A *generic-declaration* is a declaration. A *generic-declaration* is also a definition if its declaration defines a ref class, a value class, an interface class, a delegate, or a function.

A *generic-declaration* shall appear only at a namespace scope or class scope declaration.

Except for generic non-member functions, generic declarations that are also definitions can have public or private assembly visibility (§10.6.1).

A generic type shall not have the same name as any other generic type, template, class, delegate, function, object, enumeration, enumerator, namespace, or type in the same scope (C++ Standard 3.3), except as specified in 14.5.4 of the C++ Standard. Except that a generic function can be overloaded either by non-generic functions with the same name or by other generic functions with the same name, a generic name declared in namespace scope or in class scope shall be unique in that scope.

Generic type declarations follow the same rules as non-generic type declarations except where noted. Generic type declarations can be nested inside non-generic type declarations. Generic types can be nested in native classes.

Generic functions are discussed further in (§31.3).

C++/CLI permits importing from another assembly multiple generic types declared in the same scope to have the same name, provided each has a different number of generic parameters. [*Example*:

```
ref class R { … };

generic<typename T>
public ref class R { … };

generic<typename T, typename U>
public ref class R { … };
```

end example]

*using-declaration*s shall not be used to make generics from different scopes visible in a given scope, even if the generics differ in arity. Similarly, if generics from different scopes are found by a lookup because of *using-directive*s, the lookup is ambiguous.

Generics cannot be explicitly or partially specialized. [*Note*: As generics do not allow for specialization, there is no need for disambiguating names with the `typename` and `template` keywords. *end note*]

A generic function or a generic CLI class can be a friend of a native class. All specializations of a generic shall be made a friend; if any specialization of a generic is excluded from friendship, the program is ill-formed. [*Note*: As friendship is only permitted for native classes, and native classes cannot be generics, it is not possible for a generic to grant friendship to another class or function. *end note*].

31.1.1 Type parameters

A type parameter is defined in one of the following ways:

generic-parameter:
attributes$_{opt}$ `class` *identifier*
attributes$_{opt}$ `typename` *identifier*

There is no semantic difference between `class` and `typename` in a *generic-parameter*. A *generic-parameter* can optionally have one or more attributes (§29).

A *generic-parameter* defines its *identifier* to be a *type-name*.

The scope of a *generic-parameter* extends from its point of declaration until the end of the *declaration* to which its *generic-parameter-list* applies.

[*Note*: Unlike templates, generics has no equivalent to a non-type *template-parameter* or a template *template-parameter*. Neither does generics support default *generic-parameter*s; instead, generic type overloading is used. *end note*]

As a type, type parameters are purely a compile-time construct. At run-time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic type declaration. Thus, the type of a variable declared with a type parameter will, at run-time, be a closed constructed type (§31.2). The run-time execution of all statements and expressions involving type parameters uses the actual type that was supplied as the type argument for that parameter.

The literal `nullptr` cannot be converted to a type given by a generic type parameter, except if the type parameter is known to be a handle type. However, a default constructor expression can be used instead to get

a null value for a generic type parameter. In addition, a value with a type given by a generic type parameter can be compared with `nullptr` using `==` and `!=` unless the type parameter has the value type constraint (§31.4) [*Example*:

```
generic<typename T, typename U>
where U : ref class
ref class R {
  void F() {
    T t = T();     // t is initialized to default value
    U u = nullptr; // u can be initialized with nullptr,
                   // because it has the ref class constraint

    /* ... */
  }
};
```

end example]

Any type used as a generic type parameter shall have linkage.

31.1.2 Referencing a generic type by name

Like templates in Standard C++, within the body of a generic type `G<T>` any usage of the name (that is neither qualified nor a *generic-id*) of that type `G` (otherwise known as the instance type) is assumed to refer to the current instantiation. [*Example*:

```
generic<typename T>
ref class R {
public:
   R() {}      // ok: means R<T>
   void f(R^); // ok: means R<T>
   ::R g();    // error
};
```

end example]

Outside its declaration, a generic type is referenced using a constructed type (§31.2). [*Example*: Given the following,

```
generic<typename T>
ref class List {};

generic<typename U>
void f() {
   List<U>^ l1 = gcnew List<U>;
   List<int>^ l2 = gcnew List<int>;
   List<List<String^>^>^ l3 = gcnew List<List<String^>^>;
}
```

some examples of constructed types are `List<U>`, `List<int>`, and `List<List<String^>^>`. A constructed type that uses one or more type parameters, such as `List<U>`, is an open constructed type (§31.2.1). A constructed type that uses no type parameters, such as `List<int>`, is called a closed constructed type (§31.2.1). *end example*]

31.1.3 The instance type

Each type declaration has an associated constructed type, the ***instance type***. For a generic type declaration, the instance type is formed by creating a constructed type (§31.2) from the type declaration, with each of the supplied type arguments being the corresponding type parameter. Since the instance type uses the type parameters, it can only be used where the type parameters are in scope; that is, inside the type declaration. Inside the declaration of a ref class, `this` is a handle to the instance type. Inside the declaration of a value class, `this` is an interior pointer to the instance type. For non-generic types, the instance type is simply the declared type. [*Example*: The following shows several class definitions along with their instance types:

```
generic<typename T>
ref class A {        // instance type: A<T>
   ref class B {};         // instance type: A<T>::B
   generic<typename U>
   ref class C {};   // instance type: A<T>::C<U>
};

class D {};          // instance type: D
```

end example]

31.1.4 Base classes and interfaces

The base class and interfaces of a generic type declaration shall not be a type parameter, though they can be a constructed type using a type parameter. [*Example*:

```
ref class B1 {};

generic<typename T>
ref class B2 {};

generic<typename T>
interface class I1 {};

generic<typename T>
ref class R1 : T {};                // error

generic<typename T>
ref class R2 : B1 {};               // ok

generic<typename T>
ref class R3 : B2<int>, I1<int> {}; // ok (closed constructed types)

generic<typename T>
ref class R4 : B2<T>, I1<T> {};     // ok (open constructed types)
```

end example]

A generic class definition shall not use `System::Attribute` as a direct or indirect base class.

A generic class definition shall not have an indirect base class that is a template parameter.

31.1.5 Class members

All members of a generic type can use type parameters from any enclosing type, either directly or as part of a constructed type. When a particular closed constructed type (§31.1.2) is used at run-time, each use of a type parameter is replaced with the actual type argument supplied to the constructed type.

Properties, events, constructors, destructors, and finalizers shall not themselves have explicit type parameters (although they can occur in generic classes, and use the type parameters from an enclosing class).

When the type of a member is a type parameter, the declaration of that member shall use that type parameter's name without any pointer, reference, or handle declarators. Member access on a member whose type is a type parameter shall use the `->` operator. [*Example*:

```
interface class I1 {
   void F();
};
generic<typename T>
   where T : I1
ref class A {
   T t;        // no *, &, %, or ^ declarator allowed
public:
   void F() {}
   void G() {
      t->F();  // -> must be used, not .
   }
};
```

end example]

[*Note*: The compiler only generates one definition for a generic class in metadata. Generics allow value classes as generic type parameters. Textual substitution of a value class parameter would lead to an ill-formed program as the -> operator is not allowed for member access. As the VES is responsible for instantiations of generics, textual substitution is the wrong way of thinking about generic instantiation. *end note*]

As a member whose type is a parameter type will be a value class, or a handle to a ref class, interface class, delegate, or CLI array, the destructor of a generic class will not invoke the destructor on such a member.

Within a generic class definition, access to inherited protected instance members is permitted through an instance of any open constructed class type constructed from that generic class. [*Example*: In the following code

```
generic<typename T>
ref class B {
protected:
   T x;
};

generic<typename T>
ref class D : B<T> {
   static void F() {
      D<T>^ dt = gcnew D<T>;
      dt->x = T();            // Ok

      D<int>^ di = gcnew D<int>;
      di->x = 123;            // error

      D<String^>^ ds = gcnew D<String^>;
      ds->x = "test";         // error
   }
};
```

the first assignment to `x` is permitted because it takes place through an instance of an open constructed class types constructed from the generic type. However, the second and third assignments are prohibited because they take place through an instance of a closed constructed class type. When accessing members of a closed constructed generic, even within the generic definition, the access rules shall treat that class as an unrelated entity. *end example*]

Static operators are discussed in (§31.1.7), other static members are discussed in (§31.1.6), nested types are discussed in (§31.1.10), and generic functions, in general, are discussed in (§31.3).

31.1.6 Static members

A static data member in a generic class definition is shared amongst all instances of the same closed constructed type (§31.1.2), but is not shared amongst instances of different closed constructed types. These rules apply regardless of whether the type of the static data member involves any type parameters or not.

A static constructor in a generic class is used to initialize static data members and to perform other initialization for each different closed constructed type that is created from that generic class definition. The type parameters of the generic type declaration are in scope, and can be used, within the body of the static constructor.

A new closed constructed class type is initialized the first time that either:

- An instance of the closed constructed type is created.
- Any of the static members of the closed constructed type are referenced.

To initialize a new closed constructed class type, first a new set of static data members for that particular closed constructed type is created. Each of the static data members is initialized to its default value. Next, the static data members' initializers are executed for those static fields. Finally, the static constructor is executed. [*Example*:

```
generic<typename T>
ref class C {
   static int count = 0;
public:
   static C() {
      Console::WriteLine(<C<T>>::typeid);
   }

   C() {
      count++;
   }

   static property int Count {
      int get() { return count; }
   }
};

int main() {

   C<int>^ x1 = gcnew C<int>;
   Console::WriteLine(C<int>::Count);

   C<double>^ x2 = gcnew C<double>;
   Console::WriteLine(C<double>::Count);
   Console::WriteLine(C<int>::Count);

   C<int>^ x3 = gcnew C<int>;
   Console::WriteLine(C<double>::Count);

   Console::WriteLine(C<int>::Count);
}
```

The output produced is:

```
C`1[System.Int32]
1
C`1[System.Double]
1
1
1
2
```

end example]

Static operators are discussed in §31.1.7.

31.1.7 Operators

Generic class definitions can define operators and conversion functions, following the same rules as non-generic class definitions. The instance type (§31.1.3) of the class definition shall be used in the declaration of operators in accordance with the rules for operators in §19.7 or conversion functions in §14.5.3. The parameter that is not constrained by these rules can be a generic type parameter.

[*Example*: The following shows some examples of valid operator declarations in a generic class:

```
generic <typename T>
public ref struct R
{
   static R^ operator ++(R^ operand) { … }
   static int operator *(R^ op1, T op2) { … }
   static explicit operator R^(T value) { … }
};
```

end example]

31.1.8 Member overloading

Functions, instance constructors, and static operators within a generic class definition can be overloaded; however, this can lead to an ambiguity for some closed constructed types. [*Example*:

```
generic<typename T1, typename T2>
ref class X {
public:
   void F(T1, T2) { }
   void F(T2, T1) { }
   void F(int, String^) { }
};

int main() {
   X<int, double>^ x1 = gcnew X<int, double>;
   x1->F(10, 20.5);                              // okay

   X<double, int>^ x2 = gcnew X<double, int>;
   x2->F(20.5, 10);                              // okay

   X<int, int>^ x3 = gcnew X<int, int>;
   x3->F(10, 20);                                // error, ambiguous

   X<int, String^>^ x4 = gcnew X<int, String^>;
   x4->F(10, "abc");                             // error, ambiguous

}
```

end example]

A generic class is allowed to have this potential ambiguity; however, a program is ill-formed if it uses a constructed type to create such an ambiguity.

31.1.9 Member overriding

Function members in generic classes can override function members in base classes, as usual. If the base class is a non-generic type or a closed constructed type, then any overriding function member cannot have constituent types that involve type parameters. However, if the base class is an open constructed type, then an overriding function member can use type parameters in its declaration. When determining the overridden base member, the members of the base classes shall be determined by substituting type arguments, as described in §31.2.4. Once the members of the base classes are determined, the rules for overriding are the same as for non-generic classes. [*Example*:

```
generic<typename T>
ref class C abstract {
public:
   virtual T F() { … }
   virtual C<T>^ G() { … }
   virtual void H(C<T>^ x) { … }
};

ref class D : C<String^> {
public:
   virtual String^ F() override { … }     // Ok
   virtual C<String^>^ G() override { … } // Ok
   virtual void H(C<int>^ x) override { … }  // Error, should be
C<String^>
};

generic<typename T, typename U>
ref class E : C<U> {
public:
   virtual U F() override { … }           // Ok
   virtual C<U>^ G() override { … }       // Ok
   virtual void H(C<T>^ x) override { … } // Error, should be C<U>
};
```

end example]

31.1.10 Nested types

A generic class definition can contain nested type declarations, except that a generic class definition shall not contain a native class. The type parameters of the enclosing class can be used within the nested types. A nested type declaration can contain additional type parameters that apply only to the nested type. A generic type can be nested within a non-generic type.

Every type declaration contained within a generic class definition is implicitly a generic type declaration. When writing a reference to a type nested within a generic type, the containing constructed type, including its type arguments, shall be named. However, from within the outer class, the nested type can be used without qualification; the instance type of the outer class can be implicitly used when constructing the nested type. [*Example*: The following example shows three different correct ways to refer to a constructed type created from `Inner`; the first two are equivalent:

```
generic<typename T>
ref struct Outer {
   generic<typename U>
   ref class Inner {
   public:
      static void F(T t, U u) { }
   };

   static void F(T t) {
      Outer<T>::Inner<String^>::F(t, "abc");    // These two statements
have
      Inner<String^>::F(t, "abc");              // the same effect
      Outer<int>::Inner<String^>::F(3, "abc");  // This type is different
   }
};
```

end example]

A type parameter in a nested type can hide a member or type parameter declared in the outer type. [*Example*:

```
generic<typename T>
ref class Outer {
   generic<typename T>  // Valid, hides Outer's T
   ref class Inner {
      T t;              // Refers to Inner's T
   };
};
```

end example]

A program having a generic type nested within a class template is ill-formed.

31.2 Constructed types

A generic type declaration is used as a blueprint to form many different types, by way of applying type arguments (§31.2.1). A type that is named with at least one type argument is called a ***constructed type***. A constructed type can be open or closed, as we shall see in §31.2.1.

To accommodate the addition of generics, the grammar for *unqualified-id* in the C++ Standard (§5.1) is augmented, as follows by adding *generic-id*:

unqualified-id:
identifier
operator-function-id
conversion-function-id
`~` *class-name*
`!` *class-name*
template-id
generic-id
`default`

A constructed type is referred to by a *generic-id*:

generic-id:
generic-name `<` *generic-argument-list* `>`

generic-name:
identifier
operator-function-id

generic-argument-list is discussed in (§31.2.2).

31.2.1 Open and closed constructed types

All types can be classified as either ***open constructed types*** or ***closed constructed types***. An open constructed type is a type that involves type parameters. More specifically:

- A type parameter defines an open constructed type.
- A CLI array type is an open constructed type if and only if its element type is an open constructed type.
- A constructed type is an open constructed type if and only if one or more of its type arguments is an open constructed type. A constructed nested type is an open constructed type if and only if one or more of its type arguments (§31.2.2) or the type arguments of its containing type(s) is an open constructed type.

A closed constructed type is a type that is not an open constructed type.

[*Example*: Given the following,

```
generic<typename T>
ref class List {};

generic<typename U>
void f() {
   List<U>^ l1 = gcnew List<U>;
   List<int>^ l2 = gcnew List<int>;
   List<List<String^>^>^ l3 = gcnew List<List<String^>^>;
}
```

`List<U>`, `List<int>`, and `List<List<String^>^>` are examples of constructed types, where `List<U>` is an open constructed type, and `List<int>` and `List<List<String^>^>` are closed constructed types. *end example*]

At run-time, all of the code within a generic type declaration is executed in the context of a closed constructed type that was created by applying type arguments to the generic declaration. Each type parameter within the generic type is bound to a particular run-time type. The run-time processing of all statements and expressions always occurs with closed constructed types, and open constructed types occur only during compile-time processing.

Each closed constructed type has its own set of static variables, which are not shared with any other closed constructed types. Since an open constructed type does not exist at run-time, there are no static variables associated with an open constructed type. Two closed constructed types are the same type if they are constructed from the same type declaration, and their corresponding type arguments are the same type.

A constructed type has the same accessibility as its least accessible type argument.

31.2.2 Type arguments

A generic type or function is instantiated from a generic declaration by specifying type arguments that correspond to that generic declaration's type parameters. Type arguments are specified via a *generic-argument-list*:

generic-argument-list:
    generic-argument
    generic-argument-list **,** *generic-argument*

generic-argument:
    type-id

The arguments for an instantiation of a generic class shall always be explicitly specified. The arguments for an instantiation of a generic function (§31.3) can either be specified explicitly, or they can be determined by type deduction.

A *generic-argument* shall be a constructed type that is a value class, a handle to a ref class, a handle to a delegate, a handle to an interface, a handle to a CLI array, or it shall be a type parameter from an enclosing generic. [*Note*: It is not possible to use a native class, a pointer, a reference, a handle to a value class, a boxed value type, or a ref class by value as a generic argument. *end note*]

Each *generic-argument* shall satisfy any constraints (§31.4) on the corresponding type parameter.

31.2.3 Base classes and interfaces

A constructed class type has a direct base class. If the generic class definition does not specify a base class, the base class is `System::Object`. If a base class is specified in the generic class definition, the base class of the constructed type is obtained by substituting, for each *generic-parameter* in the base class definition, the corresponding *generic-argument* of the constructed type. [*Example*: Given the generic class definitions

```
generic<typename T, typename U>
ref class B { … };

generic<typename T>
ref class D : B<String^, array<T>> { … };
```

the base class of the constructed type `D<int>` would be `B<String^, array<int>>`. *end example*]

Similarly, constructed ref class, value class, and interface types have a set of explicit base interfaces. The explicit base interfaces are formed by taking the explicit base interface definitions on the generic type declaration, and substituting, for each *generic-parameter* in the base interface definition, the corresponding *generic-argument* of the constructed type.

The set of all base classes and base interfaces for a type is formed, as usual, by recursively getting the base classes and interfaces of the immediate base classes and interfaces. [*Example*: For example, given the generic class definitions:

```
ref class A { … };

generic<typename T>
ref class B : A { … };

generic<typename T>
ref class C : B<IComparable<T>^> { … };

generic<typename T>
ref class D : C<array<T>> { … };
```

the base classes of `D<int>` are `C<array<int>>`, `B<IComparable<array<int>^>>`, `A`, and `System::Object`. *end example*]

31.2.4 Class members

The non-inherited members of a constructed type are obtained by substituting, for each *generic-parameter* in the member declaration, the corresponding *generic-argument* of the constructed type. The substitution process is based on the semantic meaning of type declarations, and is not simply textual substitution (§31.1.5).

[*Example*: Given the generic class definition

```
generic<typename T, typename U>
ref class X {
   array<T>^ a;
   void G(int i, T t, X<U,T> gt);
   property U P { U get(); void set(U value); }
   int H(double d);
};
```

the constructed type `X<int, bool>` has the following members:

```
array<int>^ a;
void G(int i, int t, X<int,bool>^ gt);
property bool P { bool get(); void set(bool value); }
int H(double d);
```

end example]

The inherited members of a constructed type are obtained in a similar way. First, all the members of the immediate base class are determined. If the base class is itself a constructed type, this might involve a recursive application of the current rule. Then, each of the inherited members is transformed by substituting, for each *generic-parameter* in the member declaration, the corresponding *generic-argument* of the constructed type. [*Example*:

```
generic<typename U>
ref class B {
public:
   U F(long index);
};

generic<typename T>
ref class D : B<array<T>^> {
public:
   T G(String^ s);
};
```

In the above example, the constructed type `D<int>` has a non-inherited member `int G(String^ s)` obtained by substituting the type argument `int` for the type parameter `T`. `D<int>` also has an inherited member from the class definition `B`. This inherited member is determined by first determining the members of the constructed type `B<array<T>^>` by substituting `array<T>^` for `U`, yielding `array<T>^ F(long index)`. Then, the type argument `int` is substituted for the type parameter `T`, yielding the inherited member `array<int>^ F(long index)`. *end example*]

31.2.5 Accessibility

A constructed type `C<T1, ...,TN>` is accessible when all its parts `C`, `T1`, ..., `TN` are accessible. For instance, if the generic type name `C` is `public` and all of the *generic-arguments* `T1`, ...,`TN` are accessible as `public`, then the constructed type is accessible as `public`, but if either the type name `C` or any of the *generic-arguments* has accessibility `private` then the accessibility of the constructed type is `private`. If one *generic-argument* has accessibility `protected`, and another has accessibility `private protected`, then the constructed type is accessible only in this class and its subclasses in this assembly.

The accessibility domain for a constructed type is the most restrictive access of the open type and its type arguments. Accessibility rules for instantiations of generics are the same as for templates.

31.3 Generic functions

Member functions and non-member functions can be declared generic (§31.1). When a generic function is declared inside a ref class, value class, or interface definition, the enclosing type can itself be either generic or non-generic. If a generic function is declared inside a generic type declaration, the body of the function can refer to both the type parameters of the function, and the type parameters of the containing declaration. Not all generic type parameters to a generic function need appear as a parameter type or return type of that function. [*Example*:

```
generic<typename T>
void f1(T);

ref class C1 {
   generic<typename T, typename U>
   T f2(T t) {
      U u;
      …
   }

generic<typename T>
   T f2(T);
};
```

```
generic<typename T1>
ref class C2 {
   generic<typename T2>
   void f3(T1, array<T2>^);
};
```

end example]

Types not used as a parameter type to a generic function cannot be deduced. Types that cannot be deduced for function templates cannot be deduced for generic functions.

When used with a generic function, `static`, `extern`, and `inline` have the same meaning as when used with a non-generic function in the same context.

When the type of a parameter or variable is a type parameter, the declaration of that parameter or variable shall use that type parameter's name without any pointer, native reference, or handle declarators. [*Note*: A parameter or variable type is permitted to be a tracking reference to a type parameter. *end note*] Member access on a parameter or variable whose type is a type parameter shall use the `->` operator. [Example:

```
interface class I1 {
   void F();
};
generic<typename T>
   where T : I1
void H(T t1) {    // no *, &, or ^ declarator allowed
   T t2 = t1;     //    "     "     "     "     "
   t1->F();       // -> must be used, not .
   t2->F();       //    "     "     "
}
```

end example]

Type parameters can be used in the type of a parameter array.

A generic function can be bound to a suitably typed delegate.

31.3.1 Function signature matching rules

For the purposes of signature comparisons in function overloading, any *constraint-clause-list*s are ignored, as are the names of the function's *generic-parameter*s; however, the number of generic type parameters is relevant. [*Example*:

```
ref class A {};
ref class B {};
interface class IX {
   generic<typename T>
      where T : A
   void F1(T t);
   generic<typename T>
      where T : B
   void F1(T t);        // error, constraints are ignored
   generic<typename T>
   T F2(T t, int i);
   generic<typename U>
   void F2(U u, int i);    // error, parameter names and return
                           // type are ignored
   void F3(int x);         // no type parameters
   generic<typename T>
   void F3(int x);         // okay, different type parameter count
   generic<typename T, typename U>
   void F3(int x);         // okay, different type parameter count
   generic<typename U, typename T>
   void F3(int x);         // error, type parameter names are ignored
};
```

end example]

Functions can be overloaded; however, this can lead to an ambiguity for certain calls. [*Example*:

```
generic<typename T1, typename T2>
void F(T1, T2) { }

generic<typename T1, typename T2>
void F(T2, T1) { }

int main() {
   F<int, double>(10, 20.5);  // okay
   F<double, int>(20.5, 10);  // okay
   F<int, int>(10, 20);       // error, ambiguous
}
```

end example]

Although a program is permitted to have generic function declarations that could lead to such ambiguities, that program is ill-formed if it uses function calls to create such an ambiguity.

Generic functions can be declared `abstract`, `virtual`, and `override`. The signature matching rules described above are used when matching functions for overriding or interface implementation. When a generic function overrides a generic function declared in a base class, or implements a function in a base interface, the constraints given for each function type parameter shall be the same in both declarations. [*Example*:

```
ref struct B abstract {
   generic<typename T, typename U>
   virtual T F(T t, U u) abstract;

   generic<typename T>
      where T : IComparable
   virtual T G(T t) abstract;
};

ref struct D : B {
   generic<typename X, typename Y>
   virtual X F(X x, Y y) override;  // Okay

   generic<typename T>
   virtual T G(T t) override;       // error, constraint mismatch
};
```

The override of `F` is valid because type parameter names are permitted to differ. The override of `G` is invalid because the given type parameter constraints (in this case none) do not match those of the function being overridden. *end example*]

31.3.2 Type deduction

A call to a generic function can explicitly specify a type argument list via a *generic-id*, or it can omit that type argument list using a *generic-name* only and rely on ***type deduction*** to determine the type arguments. [*Example*:

```
ref struct X {
   generic<typename T>
   static void F(T t) {
      Console::WriteLine("one");
   }

   generic<typename T>
   static void F(T t1, T t2) {
      Console::WriteLine("two");
   }

   generic<typename T>
   static void F(T t1, int t2) {
      Console::WriteLine("three");
   }
};
```

```
int main() {
   X::F<int>(1);           // explicit, prints "one"
   X::F(1);                // deduced,  prints "one"

   X::F<double>(5.0, 6.0); // explicit, prints "two"
   X::F(5.0, 6.0);         // deduced,  prints "two"

   X::F<double>(5.0, 3);   // explicit, prints "three"
   X::F(5.0, 3);           // deduced,  prints "three"

   X::F<int>(1, 2);        // error, ambiguous
   X::F(1, 2);             // error, ambiguous
   X::F<double>(1, 2);     // explicit, prints "three"
}
```

end example] [*Example*:

```
interface class IX {};

ref class R : IX {};

generic<typename T>
void f(T) {}

void g(R^ hR) {
   f<IX^>(hR); // T is specified to be IX
   f(hR);      // T is deduced to be R
}
```

end example]

Type deduction allows a more convenient syntax to be used for calling a generic function, and allows the programmer to avoid specifying redundant type information.

In a generic function, if the type of the corresponding argument of the call is either <narrow-string-literal-type> or <wide-string-literal-type>, the deduced type, `P`, is `System::String^`. [*Note*: Type deduction on a string literal for a function template results in an array of characters instead of `System::String^`. *end note*] Otherwise, type deduction within generics is handled like type deduction within templates (C++ Standard §14.8.2).

If the generic function was declared with a parameter array, then type deduction is first performed against the function using its exact signature. If type deduction succeeds, and the resultant function is applicable, then the function is eligible for overload resolution in its normal form. Otherwise, type deduction is performed against the function in its expanded form.

An instance of a delegate can be created that refers to a generic function declaration. The type arguments used when invoking a generic function through a delegate are determined when the delegate is instantiated. The type arguments for a generic delegate can be deduced when invoking the delegate in the same manner as type deduction for invoking a generic function. If type deduction is used, the parameter types of the delegate are used as argument types in the deduction process. The return type of the delegate is not used for deduction. [Example: The following example shows both ways of supplying a type argument to a delegate instantiation expression:

```
delegate int D(String^ s, int i);
delegate int E();

ref class X {
public:
   generic<typename T>
   static T F(String^ s, T t);

   generic<typename T>
   static T G();
};

int main() {
   D^ d1 = gcnew D(X::F<int>);// okay, type argument given explicitly
   D^ d2 = gcnew D(X::F);     // okay, int deduced as type argument
   E^ e1 = gcnew E(X::G<int>);// okay, type argument given explicitly
   E^ e2 = gcnew E(X::G);     // error, cannot deduce from return type
}
```

end example]

A non-generic delegate type can be instantiated using a generic function. It is also possible to create an instance of a constructed delegate type using a generic function. In all cases, type arguments are given or deduced when the delegate instance is created, and a type-argument-list shall not be supplied when that delegate is invoked.

31.4 Constraints

The set of type arguments that is permitted for any given type parameter in a generic type or function declaration can be restricted via the use of one or more constraints. Such constraints are specified via a *constraint-clause-list*:

constraint-clause-list:
constraint-clause-list$_{opt}$ *constraint-clause*

constraint-clause:
`where` *identifier* `:` *constraint-item-list*

constaint-item-list:
constraint-item
constraint-item-list `,` *constraint-item*

constraint-item:
type-id
`ref class`
`ref struct`
`value class`
`value struct`
`gcnew ( )`

Each *constraint-clause* consists of the token `where`, followed by an *identifier* that shall be the name of a type parameter in the generic type declaration to which this *constraint-clause* applies, followed by a colon and the list of constraints for that type parameter. There shall be no more than one *constraint-clause* for each type parameter in any generic declaration, and the *constraint-clause*s can be listed in any order. The token `where` is not a keyword.

Generic constraints for generic functions are checked after overload resolution. Constraints do not influence overload resolution.

[*Note*: Because `value class` and `value struct` are turned into a single token early in the phases of translation, the following code unambiguously has the value class constraint on `T`:

```
generic<typename T>
where T : value class
V F(T t) {…}
```

It is not possible to create a constraint on a type named `value` followed by a function that uses an *elaborated-type-specifier* for a native class as a return type. *end note*]

If the type specified by *type-id* is a ref class type, it is a ***class constraint***. A class constraint shall not be `sealed`. A *constraint-item-list* shall contain no more than one class constraint.

If the type specified by *type-id* is an interface class type, it is an ***interface constraint***. The same interface type shall not be specified more than once in a given *constraint-clause*.

If the type specified by *type-id* is a generic type parameter, it is a ***naked type parameter constraint***. The same naked type parameter shall not be specified more than once in a given *constraint-clause*. A program is ill-formed if a type parameter results in a constraint upon itself, either directly or indirectly. None of the constraints specified by a naked type parameter shall conflict with other constraints given in a constraint-clause. For example, a constraint list shall not have a class constraint and a naked type parameter constraint that itself has a class constraint.

A class or interface constraint can involve any of the type parameters of the associated type or function declaration as part of a constructed type, and can involve the type being declared.

Any class or interface type specified as a type parameter constraint shall be at least as accessible as the generic type or function being declared.

If the type specified by *type-id* is anything else, the program is ill-formed.

[*Example*: The following are examples of constraints:

```
generic<typename T>
interface class IComparable {
   int CompareTo(T value);
};

generic<typename T>
interface class IKeyProvider {
   T GetKey();
};

generic<typename T>
   where T : IPrintable
ref class Printer { … };

generic<typename T>
   where T : IComparable<T>
ref class SortedList { … };

generic<typename K, typename V>
   where K : IComparable<K>
   where V : IPrintable, IKeyProvider<K>
ref class Dictionary { … };
```

end example]

If a type parameter has no constraints associated with it then it is implicitly constrained by `System::Object`. [*Note*: having a type parameter constrained in this manner severely limits what you can do with the type within the body of the generic. *end note*]

Generic *constraint-item*s shall not have an *elaborated-type-specifier*.

Constraints on generic type parameters do not have influence on the ordering or on overload resolution. The rules for partial ordering of function templates apply to generic functions.

A program that attempts to explicitly specialize a generic function using function template, is ill-formed.

31.4.1 Satisfying constraints

Whenever a constructed type or generic function is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or function. For each `where` clause, the type argument `A` that corresponds to the named type parameter is checked against each constraint as follows:

- If the constraint is a class type, an interface type, or a type parameter, let `C` represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it shall be the case that an object of type `A` is convertible to an object of type `C` by one of the following:
  - An identity conversion
  - A handle conversion
  - A boxing conversion

  [*Example*:

```
interface class I {};
ref class C : I {};
value class V : I {};

generic<typename T>
where T : I
ref class R {};

R<IF^> r1; // satisfies constraint with identity conversion
R<C^> r2;  // satisfies constraint with handle conversion
R<V> r3;   // satisfies constraint with boxing conversion

generic<typename U>
where U : T
ref class Q {
  R<U> r4; // satisfies constraint, the synthesized type for
           // U has valid conversions to T's constraint
};
```

end example]

- If the constraint is the ref class constraint, the type `A` shall satisfy one of the following:
  - `A` is a handle type.
  - `A` is a type parameter that satisfies the ref class constraint (either directly or transitively because it is constrained by another type parameter that satisfies the ref class constraint).
- If the constraint is the value class constraint, the type `A` shall satisfy one of the following:
  - `A` is a value type other than a pointer and is not the generic `System::Nullable` type. [*Note*: Note that `System::ValueType` and `System::Enum` are reference types so they do not satisfy this constraint. *end note*]
  - `A` is a type parameter having the value type constraint (either directly or transitively because it is constrained by another type parameter that has the value type constraint).
- If the constraint is the constructor constraint `gcnew()`, the type argument `A` shall not be abstract and shall have a public default constructor. This is satisfied if one of the following is true:
  - `A` is a value type, since all value types have a public default constructor.
  - `A` is a type parameter having the value type constraint.
  - `A` is a class that is not abstract, `A` contains an explicitly declared public default constructor.
  - `A` is not abstract and has a default constructor.
  - `A` is a type parameter having the constructor constraint (either directly or transitively because it is constrained by another type parameter that satisfies the constructor constraint).

A program is ill-formed if it contains a generic type one or more of whose type parameters' constraints are not satisfied by the given type arguments.

Since type parameters are not inherited, constraints are never inherited either. [*Example*: In the code below, `D` shall specify a constraint on its type parameter `T`, so that `T` satisfies the constraint imposed by the base class `B<T>`. In contrast, class `E` need not specify a constraint, because `List<T>` implements `IEnumerable` for any `T`.

```
generic<typename T>
   where T: IEnumerable
ref class B { … };

generic<typename T>
   where T: IEnumerable
ref class D : B<T> { … };

generic<typename T>
ref class E : B<List<T>^> { … };
```

end example]

31.4.2 Member lookup on type parameters

Templates wait to perform lookup with a type parameter until the type parameter is replaced by a type argument. Generics perform lookup at the point of defining the generic rather than the point of specialization. The results of lookup involving a type given by a type parameter `T` depends on the constraints, if any, specified for `T`. Lookup replaces the type of the generic type parameter `T` with a type as specified by one of the following cases:

1. If `T` has a naked type parameter constraint `N`, then a type is synthesized for `N` according to constraints and the rules one two through six below. If the synthesized type for `N` would satisfy all other constraints of `T`, then the type synthesized for `N` replaces `T`. Otherwise, all the constraints of `N` are added to the constraints of `T` and type is synthesized according to rules two through six below.
2. If `T` has no constraints or only the constructor constraint, `System::Object` replaces `T`. If lookup selects the constructor, the type is created by calling `System::Activator::CreateInstance`.
3. If `T` has the value class constraint, then a value class `V` is synthesized with the following characteristics. `V` replaces `T` for the purpose of lookup.
   - If `T` has any interface constraints, `V` provides an implementation for each interface. If lookup and overload resolution selects one of these functions, the constraint is met by the interface function implemented by the synthesized function.
4. If `T` has the ref class constraint, then a ref class `R` is synthesized with the following characteristics. `R` replaces `T` for the purpose of lookup.
   - If `T` has any interface constraints, `R` provides an implementation for each interface. If lookup and overload resolution selects one of these functions, the constraint is met by the interface function implemented by the synthesized function.
   - If `T` has the constructor constraint, `R` provides a public constructor with no parameters. If lookup selects this synthesized constructor, the type is created by calling `System::Activator::CreateInstance`.
5. If `T` has a base class constraint `B`, and if `B` would satisfy all other constraints of `T`, then `B` replaces `T`. Otherwise, a ref class `R` immediately deriving from `B` is synthesized with the following characteristics. `R` replaces `T` for the purpose of lookup.
   - If `T` has any interface constraints, `R` provides an implementation for each interface function that would not already be satisfied by deriving from `B`. If lookup and overload resolution selects one of the synthesized functions, the constraint is met by the interface function implemented by the synthesized function. [*Note*: if a base class constraint and an interface constraint has the same function signature, such that the base class function could implement the interface function, the call to that function through the generic type parameter is made through the base class constraint. *end note*]
   - If `T` has the constructor constraint, `R` provides a public constructor with no parameters. If lookup selects this synthesized constructor, the type is created by calling `System::Activator::CreateInstance`.
6. If `T` has neither a ref class constraint, a value class constraint, nor a base class constraint, a class type `RV` that is both a ref class and a value class is synthesized with the following characteristics. (Such a hybrid class can be synthesized by doing lookup twice using both a ref class and value class and ensuring that the result matches.)
   - If `T` has any interface constraints, `RV` provides an implementation for each interface. If lookup and overload resolution selects one of these functions, the constraint is met by the interface function implemented by the synthesized function.

- If `T` has the constructor constraint, the ref class represented by `RV` provides a public constructor with no parameters. If lookup selects this synthesized constructor, the type is created by calling `System::Activator::CreateInstance`.

[*Example*: Consider the following code:

```
interface class IMethod {
  void F();
};

ref struct R : IMethod {
   virtual void G() = IMethod::F {
      Console::WriteLine("R::G");
   }

   void F() {
      Console::WriteLine("R::F");
   }
};

generic<typename X>
where X : IMethod
void G1(X x) {
   x->F();
}

generic<typename X>
where X : R, IMethod
void G2(X x) {
   x->F();
}

template<typename X>
void T(X x) {
   x->F();
}

int main() {
   R^ r = gcnew R;
   G1(r);
   G2(r);
   T(r);
}
```

The program prints the following output.

```
R::G
R::F
R::F
```

`G1`'s type parameter only has one interface constraint, so a synthesized type is created with the function `F` that implements the constraint. Thus the call to `F` in the body of `G1` is through the interface. `G2`'s type parameter has both a base class constraint and an interface constraint. The base class already implements the interface, and thus `X` is replaced with the `R` within the body of `G2` for the purpose of lookup. *end example*]

31.4.3 Type parameters and boxing

When a value class type overrides a virtual method inherited from `System::Object` (such as `Equals`, `GetHashCode`, or `ToString`), invocation of the virtual function through an instance of the value class type doesn't cause boxing to occur. This is true even when the value class is used as a type parameter and the invocation occurs through an instance of the type parameter type.

Boxing never implicitly occurs when accessing a member on a constrained type parameter. For example, suppose an interface `ICounter` contains a function `Increment` which can be used to modify a value. If `ICounter` is used as a constraint, the implementation of the `Increment` function is called with a reference to the variable that `Increment` was called on, never a boxed copy.

31.4.4 Conversions involving type parameters

The conversions that are allowed on a type parameter `T` depend on the constraints specified for `T`.

For a generic type or function that have both class and interface constraints, type conversions defined in a class constraint are always preferred over those in an interface constraint.

32. Standard C and C++ libraries

Except for those requirements described elsewhere in this Standard, the interaction between the CLI library and the Standard C and C++ libraries is unspecified.

33. CLI libraries

33.1 Custom modifiers

Implementations of Standard C++ distinguish between different signatures by using name mangling; however, not only is this a language-specific solution, the mangling scheme used varies from one implementation to the next. As such, this approach is not viable in C++/CLI, where interoperability between different C++ implementations is required, and interoperability between different languages is desired. Custom modifiers address this issue.

Custom modifiers (CLI Standard, Partition II, "Types and signatures"), defined in ILAsm using `modreq` ("required modifier") and `modopt` ("optional modifier"), are similar to custom attributes except that custom attributes are attached to a declaration, while custom modifiers are part of that declaration's signature. Each custom modifer associates a type reference with an item in the signature. Two signatures that differ only by the addition of a custom modifier (required or optional) shall not be considered to match. Signature matching is discussed further in §33.1.1. Custom modifiers have no other effect on the operation of the VES.

33.1.1 Signature matching

Consider the following class definition:

```
public ref class X {
public:
   static void F(int* p1) { … }
   static void F(const int* p2) { … }
private:
   static int* p3;
   static const int* p4;
};
```

The signatures of these four members are recorded in metadata as follows:

```
.method public static void F(int32* p1) … { … }
.method public static void F(int32
  modopt([mscorlib]System.Runtime.CompilerServices.IsConst)* p2) … { … }
.field private static int32* p3
.field private static int32
  modopt([mscorlib]System.Runtime.CompilerServices.IsConst)* p4
```

[*Note*: Within the CLI context, the fully qualified name of a type uses dot (`.`) separators, while within a C++ context, a double colon (`::`) is used instead. *end note*]

Clearly, the two signatures for `F` differ, allowing these declarations as overloads.

Calls to these functions, and the corresponding code they generate, are as follows:

```
int* q1 = nullptr;
X::F(q1);
```

```
call void X::F(int32*)
```

```
const int* q2 = nullptr;
X::F(q2);
```

```
call void X::F(int32
modopt([mscorlib]System.Runtime.CompilerServices.IsConst)*)
```

The correct function is called by using an exactly matching signature in the `call` instruction. (If no matching signature is found at runtime, an exception of type `System::MissingMethodException` is thrown.)

Accesses to the data members are matched in a similar fashion:

```
static void F(int* p1) {
   p3 = p1;
   p4 = p1;
}
```

```
.method public static void F(int32* p1) … {
  …
  ldarg.0
  stsfld int32* X::p3
  ldarg.0
  stsfld int32
    modopt([mscorlib]System.Runtime.CompilerServices.IsConst)* X::p4
  …
}
```

```
static void F(const int* p2) {
   p4 = p2;
}
```

```
.method public static void F(int32
  modopt([mscorlib]System.Runtime.CompilerServices.IsConst)* p2) … {
  …
  ldarg.0
  stsfld int32 modopt([mscorlib]System.Runtime.CompilerServices.IsConst)*
    X::p4
  …
}
```

The fields are accessed using an exactly matching signature in the `stsfld` instruction. (If no matching signature is found at runtime, an exception of type `System::MissingFieldException` is thrown.)

33.1.2 modreq vs. modopt

The distinction between required and optional modifiers is important to tools (such as compilers) that deal with metadata. A required modifier indicates that there is a special semantic to the modified item, which shall not be ignored, while an optional modifier can simply be ignored. For example, `volatile`-qualified data members shall be marked with the `IsVolatile` modreq. The presence of this modifier cannot be ignored, as all accesses of such members shall involve the use of the `volatile.` prefixed instruction (see §33.1.5.9 for an example). On the other hand, the `const` qualifier can be modelled with a modopt since a `const`-qualified data member or a parameter that is a pointer to a `const`-qualified object, requires no special treatment.

The CLI itself treats required and optional modifiers in the same manner.

33.1.3 Modifier syntax

The following grammar is a subset of that defined by the CLI Standard for fields and methods. For expository purposes, this extract has been significantly simplified. (For the complete, non-simplified, version, refer to Partition II of the CLI Standard.)

Field:
`.field` *Type Id*

Method:
`.method` *Type MethodName* `(` *Parameters* `)` `{` *MethodBody* `}`

Parameters:
[*Param* [, *Param*]*]

Param:
Type [*Id*]

Type:

```
…
int32
Type *
Type [ ]
Type modreq ( [ AssemblyName ] NamespaceName . Id )
Type modopt ( [ AssemblyName ] NamespaceName . Id )
```

The *Id* in *Field* refers to the name of the data member. The *Id* in *Param* refers to the name of the optional function parameter; this name is not part of that function's signature. The *Id* in *Type* for a modopt and modreq refers to the name of the custom modifier type. This type shall be a non-nested ref class having public visibility. [*Note*: Typically, a modifier class is sealed and has no public members. *end note*] [*Example*: Here are some data and function member definitions, and the metadata produced for each of their declarations:

```
public ref class X {
   int f1;
   const int f2;
   const int* f3;
   const int** f4;
   const int* const* f5;

   array<int>^ f6;
   array<int*>^ f7;
   const array<int>^ f8;
   array<const int>^ f9;
   const int* F() { … }
   void F(int x, const int* y, array<int>^ z) { … }
};
```

```
.field private int32 f1

.field private int32
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst) f2

.field private int32
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst)* f3

.field private int32
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst)** f4

.field private int32
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst)*
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst)* f5

.field private int32[] f6

.field private int32*[] f7

.field private int32[]
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst) f8

.field private int32
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst)[] f9

.method private instance int32
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst)*
   F() … { … }

.method private instance void F(int32 x,
   int32 modopt([mscorlib]System.Runtime.CompilerServices.IsConst)*
   y, int32[] z) … { … }
```

end example]

33.1.4 Types having multiple custom modifiers

A *Type* can contain multiple modreqs and/or modopts. [*Example*:

```
public ref class X {
   const volatile int m;
};
```

```
.field private int32
   modreq([mscorlib]System.Runtime.CompilerServices.IsVolatile)
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst) m
```

end example]

33.1.5 Standard custom modifiers

With the exception of `IsVolatile` (which is defined by the CLI Standard), all of the modifiers documented in this subclause are C++/CLI-specific.

These modifier types are sealed, they are derived from `System::Object`, their public key is [00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00], they have the attribute `CLSCompliantAttribute(true)`, they belong to the library RuntimeInfrastructure, they reside in the namespace `System::Runtime::CompilerServices`, and they are part of the assembly mscorlib.

33.1.5.1 IsBoxed

[*Note*: This modreq type is *not* required by this Standard; however, at least one implementation provides it to support the handle type punctuator ^ when used with value types.

Description:

This type is used in the signature of any data member to indicate that member is a handle to a value type. It is also used in a function signature to indicate a return type and parameters that are handles to value types. When emitted, this type shall be immediately preceded by `class [mscorlib]System.ValueType` and `modopt(`*v*`)`, in that order, where *v* is the value type name.

```
public value class V {};
public ref class C {};

public ref class X {
   int* m1;
   int^ m2;
   V^   m3;
   C^   m4;

public:
   void F(int* x) { … }
   void F(int^ x) { … }
   const signed char^ F(V^ v, C^ c) { … }
};
```

```
.field private int32* m1

.field private class [mscorlib]System.ValueType
   modopt([mscorlib]System.Int32)
   modreq([mscorlib]System.Runtime.CompilerServices.IsBoxed) m2

.field private class [mscorlib]System.ValueType modopt(V)
   modreq([mscorlib]System.Runtime.CompilerServices.IsBoxed) m3

.field private class C m4

.method public instance void F(int32* x) … { … }

.method public instance void F(class [mscorlib]System.ValueType
   modopt([mscorlib]System.Int32)
   modreq([mscorlib]System.Runtime.CompilerServices.IsBoxed) x) … { … }

.method public instance class [mscorlib]System.ValueType
   modopt([mscorlib]System.Runtime.CompilerServices.IsConst)
   modopt([mscorlib]System.SByte)
   modreq([mscorlib]System.Runtime.CompilerServices.IsBoxed)
   F(class [mscorlib]System.ValueType modopt(V)
   modreq([mscorlib]System.Runtime.CompilerServices.IsBoxed) v,
   class C c) … { … }
```

In the case of `m2`, the signature indicates that the field is a handle to type `System::ValueType`. The particular kind of value type is then indicated by the value-type special modopt that follows, `[mscorlib]System.Int32`; that is, type `int`. Similarly, in the case of `m3`, this value-type special modopt

is the user-defined type `V`. The second and third overloads of `F` also use value-type special modopts, namely `[mscorlib]System.Int32` and `[mscorlib]System.SByte`, to indicate `int` and `signed char`, respectively. As suggested by this example, a value-type special modopt can be any value type. As such, `C` does not result in modopt generation, as that type is a ref type, not a value type. *end note*]

33.1.5.2 IsByValue

This modreq type supports the passing of objects of a ref class type by value.

Description:

This type is used in the signature of a function. However, it is not used to indicate that a ref class value is returned by a function; for that, see `IsUdtReturn` (§33.1.5.8). [*Example*:

```
public ref struct R {
   static void F(R r) { … }
};
```

```
.class public … R … {
  .method public static void F(class R modopt(
    [mscorlib]System.Runtime.CompilerServices.IsByValue) r) … { … }
}
```

end example]

33.1.5.3 IsConst

This modopt type supports the `const` qualifier.

Description:

This type can be used in the signature of any data member or function.

Numerous examples of the use of this modifier are shown in §33.1.1, §33.1.3, and §33.1.4.

33.1.5.4 IsExplicitlyDereferenced

This modopt type supports the use of interior pointers and pinning pointers.

Description:

This type can be used in the signature of any function or local variable. [*Example*:

```
public ref struct X {
   void F(interior_ptr<int> x) { … }
   void F(interior_ptr<unsigned char> x) { … }
};
```

```
.method … void F(int32& modopt(
  [mscorlib]System.Runtime.CompilerServices.IsExplicitlyDereferenced) x)
  … { … }

.method … F(unsigned int8& modopt(
  [mscorlib]System.Runtime.CompilerServices.IsExplicitlyDereferenced) x)
  … { … }
```

end example]

33.1.5.5 IsImplicitlyDereferenced

This modopt type supports the reference type punctuators & and %.

Description:

This type is used in the signature of any data member to indicate that member is a reference. It is also used in a function signature to indicate parameters that are passed by reference or that that function returns by reference. [*Example*:

```
ref class X {
   int* m1;
   int& m2;
public:
   void F(int* x) { … }
   void F(int& x) { … }
   void F(X% x) { … }
   int& G() { … }
};
```

```
.field private int32* m1

.field private int32* modopt(
  [mscorlib]System.Runtime.CompilerServices.IsImplicitlyDereferenced) m2

.method … void F(int32* x) … { … }

.method … void F(int32* modopt(
  [mscorlib]System.Runtime.CompilerServices.IsImplicitlyDereferenced) x)
  … { … }

.method … void  F(class X modreq([mscorlib]
   System.Runtime.CompilerServices.IsImplicitlyDereferenced) x) … { … }

.method … int32* modopt([mscorlib]
  System.Runtime.CompilerServices.IsImplicitlyDereferenced) G() … { … }
```

end example]

33.1.5.6 IsLong

[*Note*: This modopt type is *not* part of this Standard; however, it is used by at least one implementation for two unrelated purposes: supporting the types `long int` and `unsigned long int` as synonyms for `int` and `unsigned int`, respectively, and supporting the type `long double` as a synonym for `double`.

Description:

`IsLong` can be used in the signature of any data member or function.

```
public ref class X {
   int i;
   long int li;
   double d;
   long double ld;
public:
   unsigned int F(unsigned int* pu) { … }
   unsigned long int F(unsigned long int* pul) { … }

   double F(double* pd) { … }
   long double F(long double* pld) { … }
};
```

```
.field private int32 i

.field private int32
  modopt([mscorlib]System.Runtime.CompilerServices.IsLong) li

.field private float64 d

.field private float64
  modopt([mscorlib]System.Runtime.CompilerServices.IsLong) ld

.method … unsigned int32 F(unsigned int32* pu) … { … }

.method … unsigned int32
  modopt([mscorlib]System.Runtime.CompilerServices.IsLong)
  F(unsigned int32
  modopt([mscorlib]System.Runtime.CompilerServices.IsLong)* pul)
  … { … }

.method … float64 F(float64* pd) … { … }
```

```
.method … float64
  modopt([mscorlib]System.Runtime.CompilerServices.IsLong)
  F(float64 modopt([mscorlib]System.Runtime.CompilerServices.IsLong)*
  pld) … { … }
```

end note]

33.1.5.7 IsSignUnspecifiedByte

This modopt type supports plain `char`'s being a type separate from `signed char` and `unsigned char`.

Description:

This type can be used in the signature of any data member or function. [*Example*:

```
public ref class x {
   char c;
   signed char sc;
   unsigned char uc;
public:
   char* F(char* p1) { … }
   char* F(signed char* p2) { … }
   char* F(unsigned char* p2) { … }
};
```

The code generated from an implementation in which a plain `char` is signed, as as follows:

```
.field private int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte) c

.field private int8 sc

.field private unsigned int8 uc

.method … int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)*
  F(int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)* p1)
  … { … }

.method … int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)*
  F(int8* p2) … { … }

.method … int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)*
  F(unsigned int8* p2) … { … }
```

while that generated from an implementation in which a plain `char` is unsigned, is shown below:

```
.field private unsigned int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte) c

.field private int8 sc

.field private unsigned int8 uc

.method … unsigned int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)*
  F(unsigned int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)* p1)
  … { … }

.method … unsigned int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)*
  F(unsigned int8* p2) … { … }

.method … unsigned int8 modopt(
  [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)*
  F(unsigned int8* p2) … { … }
```

end example]

33.1.5.8 IsUdtReturn

This modreq type supports the returning of objects of a ref class type by value.

Description:

This type is used in the signature of a function. However, it is not used to indicate a ref class value that is passed to a function; for that, see `IsByValue` (§33.1.5.2). [*Example*:

```
public ref struct R {
   R() { … }
   R(R% r) { … }
   R F() { … }
};
```

```
.method … void modreq([mscorlib]
  System.Runtime.CompilerServices.IsUdtReturn) F(class R& A_1) … { … }
```

end example]

33.1.5.9 IsVolatile

This modreq type supports the `volatile` qualifier. (Although `IsVolatile` is part of the CLI Standard, for convenience, it is documented here as well.)

Description:

This type can be used in the signature of any data member or function.

`volatile`-qualified data member, local variable, and parameter declarations shall be marked with this modreq. Furthermore, each access to such a member, variable, or parameter shall also be marked with this modreq.

Any compiler that imports metadata having signature items that contain the `volatile` modreq is required to use `volatile.` prefixed instructions when accessing memory locations that are `volatile`-qualified. [*Example*:

```
public ref class x {
   volatile int* p1;
public:
   void F(volatile int* p2, int* p3)
   {
      *p1 = 1;
      *p2 = 2;
      *p3 = 3;
      p1 = 0;
   }
};
```

```
.field private int32
   modreq([mscorlib]System.Runtime.CompilerServices.IsVolatile)* p1

.method … void F(int32
   modreq([mscorlib]System.Runtime.CompilerServices.IsVolatile)* p2,
   int32* p3) … {
   …
  ldarg.0
  ldfld int32 modreq([mscorlib]
      System.Runtime.CompilerServices.IsVolatile)* x::p1
  ldc.i4.1
  volatile.    // prefix instruction needed when dereferencing p1
  stind.i4

  ldarg.1
  ldc.i4.2
  volatile.    // prefix instruction needed when dereferencing p2
  stind.i4

  ldarg.2
  ldc.i4.3
  stind.i4     // no prefix instruction needed when dereferencing p3
```

```
  ldarg.0
  ldc.i4.0
  stfld int32 modreq([mscorlib]
      System.Runtime.CompilerServices.IsVolatile)* x::p1
               // no prefix instruction needed; not dereferencing p1
  ret
}
```

Note that given the declaration `volatile int* p1`, `p1` is not itself `volatile`-qualified; however, `*p1` is. *end example*]

33.2 Standard attributes

A conforming C++/CLI implementation shall provide the attribute types described below:

33.2.1 NativeCppClass

Each native class is encoded in metadata as a value class marked with the attribute `NativeCppClass`, which is defined as follows:

```
[System::AttributeUsage(System::AttributeTargets::Struct,Inherited=true)]
public ref class NativeCppClassAttribute sealed : System::Attribute {
public:
   NativeCppClassAttribute () { /* … */ }
};
```

This type has the following characteristics: Its public key is [00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00], it has the attribute `CLSCompliantAttribute(true)`, it belongs to the library RuntimeInfrastructure, it resides in the namespace `System::Runtime::CompilerServices`, and it is part of the assembly mscorlib.

34. Metadata

This clause is intended to introduce metadata generation; however, the coverage is not exhaustive. For a definitive description of that topic, refer to the CLI standard, especially Partition II.

34.1 Basic concepts

34.1.1 Importing types from assemblies

Ordinarily, when types are referred to in metadata, they are fully qualified using the following form:

`[` *assembly-name* `]` *namespace-name* `.` *type-name*

Exceptions are C++/CLI fundamental type names (which are synonyms for CLI built-in type names) and synonyms for CLI built-in type names used directly. [*Example*:

```
#using <mscorlib.dll>   // redundant
#using <System.dll>     // needed for Socket
#using <System.Xml.dll> // needed for XmlTextReader

int main() {
   System::Text::StringBuilder^ strBld;
   System::Net::Sockets::Socket^ soc;
   System::Xml::XmlTextReader^ xtr;

   int i;                  // a synonym for System::Int32;
                           // which is equivalent to int32
   System::Int64 j;        // equivalent to int64
   System::String^ str;    //      "     "  string
   System::Object^ obj;    //      "     "  object
}
```

```
.method … main() … {
  …
  .locals ([0] class [mscorlib]System.Text.StringBuilder V_0,
           [1] class [System.Xml]System.Xml.XmlTextReader V_1,
           [2] class [System]System.Net.Sockets.Socket V_2,
           [3] int32 V_3,
           [4] int64 V_4,
           [5] string V_5,
           [6] object V_6)
  …
}
```

end example]

34.2 Types

34.2.1 Reference types

A tracking reference to a ref class or interface class type shall be emitted into metadata as that type with the modopt `IsImplicitlyDereferenced` (§33.1.5.5). A tracking reference to a value class type shall be emitted into metadata as a managed pointer to type without that modopt. [*Example*:

```
public ref class R {};
public value class V {};
public interface class I {};

void F1(R% tr1) {}
void F2(I% tr2) {}
void F3(V% tr3) {}
void F4(int% tr3) {}
```

```
.method assembly static void  F1(class R modreq([mscorlib]
  System.Runtime.CompilerServices.IsImplicitlyDereferenced) tr1) … { … }
.method assembly static void  F2(class I modreq([mscorlib]
  System.Runtime.CompilerServices.IsImplicitlyDereferenced) tr2) … { … }
.method assembly static void  F3(valuetype V& tr3) … { … }
.method assembly static void  F4(int32& tr3) … { … }
```

end example]

34.2.2 Interior pointers

An interior pointer to a type shall be emitted into metadata as a managed pointer to that type with the modopt `IsExplicitlyDereferenced` (§33.1.5.4). [*Example*:

```
public ref class R {};
public value class V {};
public interface class I {};

void F1(interior_ptr<R^> ip1) {}
void F2(interior_ptr<I^> ip2) {}
void F3(interior_ptr<V> ip3) {}
void F4(interior_ptr<int> ip3) {}
```

```
.method assembly static void  F1a(class R& modopt([mscorlib]
  System.Runtime.CompilerServices.IsExplicitlyDereferenced) ip1) … { … }
.method assembly static void  F2a(class I& modopt([mscorlib]
  System.Runtime.CompilerServices.IsExplicitlyDereferenced) ip2) … { … }
.method assembly static void  F3a(valuetype V& modopt([mscorlib]
  System.Runtime.CompilerServices.IsExplicitlyDereferenced) ip3) … { … }
.method assembly static void  F4a(int32& modopt([mscorlib]
  System.Runtime.CompilerServices.IsExplicitlyDereferenced) ip3) … { … }
```

end example]

34.2.3 Pinning pointers

A pinning pointer shall be emitted into metadata with the modifier `pinned` and the modopt `IsExplicitlyDereferenced` (§33.1.5.4). [*Example*:

```
value struct V {
   int Data;
   void F() {
      pin_ptr<V> ppv = this;
      V* pv = ppv;
   }
};

int main() {
   V v;
   pin_ptr<V> ppv = &v;
   int* pi = &ppv->Data;
}
```

```
.class …  V … {
  .field public int32 Data
  .method … F() … {
    …
    .locals ([0] valuetype V& pinned modopt([mscorlib]
       System.Runtime.CompilerServices.IsExplicitlyDereferenced) V_0,
             [1] valuetype V* V_1)
    …
  }
}
```

```
.method … main() … {
  …
  .locals ([0] valuetype V& pinned modopt([mscorlib]
    System.Runtime.CompilerServices.IsExplicitlyDereferenced) V_0,
          [1] int32* V_1,
          [2] valuetype V V_2)
  …
}
```

end exaple]

34.2.4 Native arrays

The encoding of native arrays in metadata is unspecified. [*Note*: This does not cause interop problems because such arrays cannot have `public` visibility. *end note*]

34.3 Variables

34.3.1 File-scope and namespace-scope variables

The encoding of file-scope and namespace-scope variable declarations and definitions in metadata is unspecified. [*Note*: This does not cause interop problems because such declarations and definitions cannot have `public` visibility. *end note*]

34.4 Conversions

34.4.1 String literal conversions

When a <narrow-string-literal-type> or <wide-string-literal-type> is converted to `System::String^`, the result is treated as a CLI string literal. [*Example*:

```
void F(String^ s);

F("red\t" "car\n");
F("ABC\xFF");
F(L"blue");
F(L"\xFF" L"\xFE");
```

```
ldstr    "red\tcar\n"
call     void F(string)

ldstr    bytearray (41 00 42 00 43 00 FF 00 )
call     void F(string)

ldstr    "blue"
call     void F(string)

ldstr    bytearray (FF 00 FE 00 )
call     void F(string)
```

end example]

34.4.2 Boxing conversions

A boxing conversion is achieved via the box instruction, as specified in the CLI Standard, Partition III, §4. This causes a runtime bitwise copy of the value class instance to an object on the CLI heap. [*Example*:

```
int main() {
   Console::WriteLine("{0}, {1}", 10, TimeSpan::MinValue);
}
```

```
.method … main() … {
  ldstr      "i = {0}"
  ldc.i4.s   10
  box        [mscorlib]System.Int32
  ldsfld     valuetype [mscorlib]System.TimeSpan
   [mscorlib]System.TimeSpan::MinValue
  box        [mscorlib]System.TimeSpan
  call       void [mscorlib]System.Console::WriteLine(string, object,
object)
  ldc.i4.0
  ret
}
```

end example]

34.4.3 Conversion functions

In ref classes, implicit conversion functions shall have the name `op_Implicit`, and explicit conversion functions shall have the name `op_Explicit`. In native classes, implicit conversion functions shall have the name `<op_Implicit>`, and explicit conversion functions shall have the name `<op_Explicit>`. All conversion functions shall be marked `specialname`. `op_Implicit` and `op_Explicit` can be overloaded on their return type. [*Example*:

```
public value struct Decimal {
   …
   static operator Decimal(int value);
   static explicit operator double(Decimal value);

   explicit operator float();
};
```

```
.class public sequential … Decimal … {
  .method public specialname static valuetype Decimal op_Implicit(
    int32 value) … { … }
  .method public specialname static float64 op_Explicit(
    valuetype Decimal value) … { … }

  .method public specialname instance float32 op_Explicit()
    … { … }
}
```

end example]

Converting constructors are emitted as constructors, never as conversion functions. (Constructors in ref classes and value classes are always explicit.)

34.5 Expressions

34.5.1 Class member access

When using an instance of a value type to call a virtual function in a base class (which can only be `System::ValueType` or `System::Object`), and that value type does not itself override that function, the instance of the value type shall be boxed. In no other cases shall accessing a member of a value type cause boxing. [*Example*:

```
value struct V {
   virtual int GetHashCode() override { … }
};

int main() {
   V v;
   … = v.GetHashCode(); // calls V::GetHashCode
   … = v.ToString();    // calls ValueType::ToString
}
```

```
.method … main() … {
  …
  .locals ([0] valuetype V V_0)
  ldloca.s   V_0
  initobj    V
  ldloca.s   V_0
  call       instance int32 V::GetHashCode()

  …
  ldloc.0
  box        V
  callvirt   instance string [mscorlib]System.ValueType::ToString()
  …
}
```

As `V` overrides `GetHashCode`, no box instruction is needed before the call instruction. However, as `V` does not override `ToString`, the version from `ValueType` is used, resulting in a box instruction followed by a callvirt instruction.

end example]

34.5.2 Dynamic cast

If a run-time check is applied to the cast, and `T` is a handle or reference to a CLI class type, the run-time check shall be performed using the isinst instruction.

34.5.3 Safe cast

When a "handle to cv2 `B`" is cast to a "handle to cv1 `D`", a run-time check is performed by the castclass instruction to determine that `D` inherits from `B`. The result of the conversion is the result of that instruction.

When a "cv2 `B`" is cast to a "tracking reference to cv1 `D`", a run-time check is performed by the castclass instruction to determine that `D` inherits from `B`. The result is the dereferenced result of castclass.

When an rvalue of type "handle to cv1 `R`" is converted to an lvalue of type `V`, the unbox instruction is used.

34.6 Functions

34.6.1 Name lookup

On input, the presence or absence of the hidebysig notation in metadata is ignored; all native types are treated as having hidebyname members while all CLI class types are treated as having hidebysig members. [*Note*: On output, CLI class types shall have each of their members marked `hidebysig` (§34.7.4). *end note*]

34.6.2 Parameter arrays

A function can have a parameter array as its final parameter only. Such a parameter shall result in a `.custom` directive for the standard attribute `System::ParamArrayAttribute`, on the final parameter in the `.method` directive generated for that function. [*Example*:

```
void f(... array<Object^>^ p) { … }

int main() {
   array<Object^>^ a1 = gcnew array<Object^>(2);
   array<Object^>^ a2 = gcnew array<Object^>(4);
   array<Object^>^ a3 = gcnew array<Object^>(8);

   f(a1);
   f(a2, a1);
   f(a1, a3, a2);
}
```

```
.method assembly static void f(object[] p) … {
  .param [1]
  .custom instance void [mscorlib]System.ParamArrayAttribute::.ctor()
    = ( 01 00 00 00 )
  …
}
```

end example]

The final parameter of a function taking a parameter array is a handle to a CLI array of the given type. Calls to such a function shall be translated into an allocation of a CLI array of the given type, with consecutive elements of that array being initialized with the arguments passed to the function, in their lexical order.

[*Example*: Here's an example of using a parameter array with a member function:

```
public ref struct C {
   static void F(int val, ... array<String^>^ list) { … }

   static void TestF() {
      F(10, "red", "blue", "green");
   }
};
```

```
.class public … C … {
    .method public static void  F(int32 val, string[] list) … {
    .param [2]
    .custom instance void [mscorlib]System.ParamArrayAttribute::.ctor()
       = ( 01 00 00 00 )
  }

  .method public static void  TestF() … {
    .maxstack  3
    .locals (string[] V_0)
    ldc.i4.3
    newarr     [mscorlib]System.String
    stloc.0
    ldloc.0
    ldc.i4.0

    ldstr      "red"
    stelem.ref
    ldloc.0
    ldc.i4.1

    ldstr      "blue"
    stelem.ref
    ldloc.0
    ldc.i4.2

    ldstr      "green"
    stelem.ref
    ldc.i4.s   10
    ldloc.0
    call       void C::F(int32, string[])
    ret
  }
}
```

end example]

34.6.3 Importing native functions

If a function has the attribute `DllImportAttribute` (in namespace `System::Runtime::InteropServices`), the compiler is required to not preserve that type in metadata as a custom attribute. Instead, the compiler shall emit it directly in the file format. (Consumers of such metadata are required to retrieve this data from the file format and return it as if it were a custom attribute.)

The `.method` directive generated shall be marked with the `pinvokeimpl` predefined attribute, whose first quoted string is a platform-specific description indicating where the implementation of the function is

located, and whose optional second string is the name of the function as it exists on that platform. The body of the method shall be empty. [*Example*:

```
// MyCLib.h
using namespace System::Runtime::InteropServices;
[DllImport("MyCLib.dll", CallingConvention =
CallingConvention::StdCall, EntryPoint="Hypot" )]
extern "C" double Hypotenuse(double s1, double s2);
```

```
.method public static pinvokeimpl("MyCLib.dll" as "Hypot" stdcall)
   float64  Hypotenuse(float64 s1, float64 s2) cil managed {}
}
```

```
// MyCLibApp.cpp
#include "MyCLib.h"

int main() {
   Console::WriteLine("Hypotenuse = {0}", Hypotenuse(3, 4));
}
```

```
.method … main() … {
  ldstr      "Hypotenuse = {0}"
  ldc.r8     3.
  ldc.r8     4.
  call       float64 Hypotenuse(float64, float64)
  box        [mscorlib]System.Double
  call       void [mscorlib]System.Console::WriteLine(string, object)
  ldc.i4.0
  ret
}
```

end example]

If a function parameter or return value has the attribute `MarshalAsAttribute` (in namespace `System::Runtime::InteropServices`), the compiler is required to not preserve that type in metadata as a custom attribute. Instead, the compiler shall emit it directly in the file format. (Consumers of such metadata are required to retrieve this data from the file format and return it as if it were a custom attribute.) The parameters or return type in the `.method` directive generated shall be marked with the `marshal` attribute according to the `UnManagedType` argument passed. [*Example*:

```
using namespace System::Runtime::InteropServices;
[DllImport("msvcrt.dll", CallingConvention = CallingConvention::Cdecl)]
extern "C" int strcmp([MarshalAs(UnmanagedType::LPStr)] System::String^
s1,
[MarshalAs(UnmanagedType::LPStr)] System::String^ s2);
```

```
.method public static pinvokeimpl("msvcrt.dll" cdecl)
   int32 strcmp(string marshal(lpstr) s1, string marshal(lpstr) s2)
   cil managed {}
```

end example]

34.6.4 Non-member functions

The encoding of non-member functions in metadata is unspecified. [*Note*: This does not cause interop problems because such functions cannot have `public` visibility. *end note*]

34.7 Classes and members

34.7.1 Class definitions

A ref class, value class, or interface class shall be emitted using a class directive having the corresponding name and visibility. It can be marked with the following:

- Any one of the "Marshal string" attributes `ansi`, `autocode`, or `unicode` (§34.7.3).
- Any one of the "Type layout" attributes `auto`, `explicit`, or `sequential` (§34.7.3).

- Any combination of the "Special handling" attributes `beforefieldinit`, `rtspecialname`, `serializable`, or `specialname`. (For more information about serialization, see the note below.)

A nested ref class or value class shall be marked `nested`, followed by the appropriate accessibility, and shall be defined inside the type in which it is nested.

A ref class shall be emitted with an `extends` clause, which specifies either the explicitly given direct base class or the default base class, `[mscorlib]System::Object`. If the class implements any interfaces, a corresponding `implements` clause shall be present.

A value class shall extend `[mscorlib]System::ValueType`, it shall have a type layout of `sequential`, and it shall be marked `sealed`.

An interface class shall be marked `interface` and `abstract`.

[*Example*:

```
public ref class B { … };

public ref struct D : B {
   ref class N { … };
};

private value struct S { … };

interface class I { … };
```

```
.class public auto ansi B extends [mscorlib]System.Object { … }

.class public auto ansi D extends B {
  .class auto ansi nested public N extends [mscorlib]System.Object { … }
}

.class private sequential ansi sealed S extends:
  [mscorlib]System.ValueType { … }

.class interface private abstract auto ansi I { … }
```

end example]

The encoded name of a class includes its parent namespaces, if any, with each pair of identifiers being separated by a period.

[*Example*:

```
namespace NS1 {
   public struct N {
      ref struct R1 { … };
   };

   namespace NS2 {
      public ref struct R2 {
         value struct V { … };
      };
   }
}
```

```
.class public sequential ansi sealed NS1.N extends
    [mscorlib]System.ValueType {
  .class auto ansi nested public R1 extends [mscorlib]System.Object { … }
}

.class public auto ansi NS1.NS2.R2 extends [mscorlib]System.Object {
  .class sequential ansi sealed nested public V extends
    [mscorlib]System.ValueType { … }
}
```

end example]

For information specific to generic types, see §34.18.

[*Note*: The CLI standard does not define the process of serialization and deserialization. However, it does make provision for such a facility by defining a metadata attribute `serializable`, which can be applied to a class definition. This attribute indicates that, by default, all the instance data members in that type should be persisted when their parent object is serialized. The CLI standard also defines a metadata attribute `notserialized`, which can be applied to an instance data member definition, to indicate that that member *not* be persisted when its parent object is serialized.

In an extended implementation, these metadata attributes might be generated, by example, by the compiler's recognizing attributes called `System::Runtime::Serialization::SerializableAttribute` and `System::Runtime::Serialization::NonSerializedAttribute`, respectively.

All of the types in the CLI standard library are *required* to have the `serializable` attribute. *end note*]

34.7.1.1 Abstract classes

A ref class explicitly declared `abstract` shall be emitted as a class marked `abstract`. [*Example*:

```
public ref struct B abstract { … };
.class public abstract … B … { … }
```

end example]

34.7.1.2 Sealed classes

A ref class explicitly declared `sealed` shall be emitted as a class marked `sealed`. All value classes shall be marked `sealed`. [*Example*:

```
public ref struct B sealed { … };

private value struct C { … };
.class public … sealed B … { … }

.class private … sealed C … { … }
```

end example]

34.7.2 Member access

Each *access-specifier* has a corresponding metadata accessibility attribute, as follows:

C++/CLI Access Specifier	Metadata Accessibility Attribute
`private`	`private`
`protected`	`family`
`public`	`public`
`internal`	`assembly`
`protected public`	`famorassem`
`public protected`	`famorassem`
`protected private`	`famandassem`
`private protected`	`famandassem`

Each member shall have its own accessibility attribute, as required. [*Example*:

```
public ref class C {
private:
   int m1;

protected:
   int m2;

public:
   int m3;
```

```
internal:
   int m4;
protected public:
   int m5;
public protected:
   int m6;
private protected:
   int m7;
protected private:
   int m8;
};
```

```
.class public … C … {
  .field private int32 m1
  .field family int32 m2
  .field public int32 m3
  .field assembly int32 m4
  .field famorassem int32 m5
  .field famorassem int32 m6
  .field famandassem int32 m7
  .field famandassem int32 m8
}
```

end example]

34.7.3 Data members

Each data member shall correspond to a field having the corresponding type and accessibility attribute. (For information about accessibility of members see §34.7.2.)

A static data member shall have the `static` attribute, while an instance data member shall not. [*Example*:

```
public ref class C {
   int count;
   float* pCoeff;
   array<long long int>^ values;
   C^ next;
   System::Exception^ lastException;
   static int objectCount;
   static String^ name;
};
```

```
.class public … C … {
  .field private int32 count
  .field private float32* pCoeff
  .field private int64[] values
  .field private class C next
  .field private class [mscorlib]System.Exception lastException
  .field private static int32 objectCount
  .field private static string name
}
```

end example]

If a static data member contains an *initializer*, the initialization of the corresponding field shall be done in the parent class's static constructor.

If a ref or value class type has the attribute `StructLayoutAttribute` (in namespace `System::Runtime::InteropServices`), the compiler is required to not preserve that type in metadata as a custom attribute. Instead, the compiler shall emit it directly in the file format. (Consumers of such metadata are required to retrieve this data from the file format and return it as if it were a custom attribute.) This attribute can be used to specify the layout of a data structure via the `auto`, `explicit`, and `sequential` attributes on the class definition, the alignment (via a `.pack` directive), the size (via a `.size` directive), and the marshalling of strings via the `ansi`, `auto`, and `unicode` attributes on the class definition.

An instance data member can have the attribute `FieldOffsetAttribute` (in namespace `System::Runtime::InteropServices`), which controls the exact placement of that member. As with the attribute `StructLayoutAttribute`, the compiler shall emit the affects of `FieldOffsetAttribute` directly in the file format, rather than emitting the attribute itself.

[*Example*:

```
using namespace System::Runtime::InteropServices;

[StructLayout(LayoutKind::Explicit)]
public value class S1 {
   [FieldOffset(0)] int v;
   [FieldOffset(4)] unsigned char c;
   [FieldOffset(8)] int w;
};
```

```
.class public explicit ansi … S1 … {
  .pack …
  .size 0
  .field [4] private unsigned int8 c
  .field [0] private int32 v
  .field [8] private int32 w
}
```

```
[StructLayout(LayoutKind::Sequential, Pack=4)]
public value class S2 {
   int v;
   unsigned char c;
   int w;
};
```

```
.class public sequential ansi … S2 … {
  .pack 4
  .size 0
  .field private unsigned int8 c
  .field private int32 v
  .field private int32 w
}
```

```
[StructLayout(LayoutKind::Explicit, Size=12, CharSet=CharSet::Unicode)]
public ref class S3 {
   [FieldOffset(0)] int* pi;
   [FieldOffset(0)] unsigned int ptrValue;
};
```

```
.class public explicit unicode S3 … {
  .pack …
  .size 12
  .field [0] private int32* pi
  .field [0] private unsigned int32 ptrValue
}
```

end example]

For information about literal and initonly fields see §34.7.11 and §34.7.12, respectively.

A field definition can optionally contain the `notserialized` attribute. (For more information about serialization, see the note in §34.7.1.)

Ordinarily, a field shall not be marked `rtspecialname` or `specialname`. However, the instance field called `value__` that is emitted in an enum's class shall be marked `rtspecialname` and `specialname`.

Data members can have applied to them the attribute `MarshalAsAttribute` (in namespace `System::Runtime::InteropServices`). For metadata information on this attribute, see §34.6.3.

34.7.4 Functions

A function shall be emitted as a `.method` directive. Ordinarily, a method definition shall not be marked `rtspecialname` or `specialname`. (Instance and static constructors are exceptions; see §34.7.9 and

§34.7.10, respectively.) The definition of a static function shall be marked `static`; that for an instance function shall be marked `instance`.

Member functions of ref classes, value classes, and interface classes shall be marked `hidebysig`.

Virtual member functions of ref classes, value classes, and interface classes shall be marked `strict`, while non-virtual member functions from those types shall not. [*Note*: The CLI requires that `strict virtual` methods can only be overridden if they are also accessible. *end note*]

Ordinarily, the name of the method emitted shall be the same as that in its source declaration; however, instance constructors (§34.7.9), static constructors (§34.7.10), property accessors (§34.7.5), event accessors (§34.7.6), and static operators (§34.7.7) are exceptions.

The return type, and the types and order of the parameters in the parameter list emitted shall correspond directly to that in the function's source declaration.

The accessibility of a function shall be reflected in the definition of its `.method` directive. (See §34.7.2.)

A method definition shall be marked with the appropriate implementation attributes, such as `cil managed` (see discussion below).

[*Example*:

```
public ref class C {
   static void compressData(int* p1, String^ p2, Object^ p3) { … }
public:
   void Initialize() { … }
   void Initilaize(int i, int j) { … }
   virtual void Display() { … }
};
```

```
.class public … C … {
  .method private hidebysig static void compressData(int32* p1,
      string p2, object p3) cil managed { … }

  .method public hidebysig instance void Initialize() cil managed { … }

  .method public hidebysig instance void Initilaize(int32 i, int32 j)
    cil managed { … }

  .method public hidebysig strict newslot virtual instance void Display()
    cil managed { … }
}
```

end example]

34.7.4.1 Override functions

Use of an *override-specifier* shall always result in an `.override` directive in the metadata, while use of the *function-modifier* `override` without an *override-specifier* shall not. [*Example*: Given the following code

```
public ref struct B {
   virtual void F() {};
   virtual void F(int i) {};
};

public ref struct D1 : B {
   virtual void F() override {}        // explicitly overrides B::F()
};

public ref struct D2 : B {
   virtual void F() override {}        // explicitly overrides B::F()
   virtual void G(int i) = B::F {}     // named override of B::F(int)
};

public ref struct D3 : B {
   virtual void F() = B::F {}          // explicitly overrides B::F()
};
```

the relevant metadata generated for classes `D2` and `D3` is as follows:

```
.class public … D2 extends B {
  .method public virtual instance void F() … {
  …
  }

  .method public newslot virtual final instance void G(int32 i) … {
    .override B::F      // overrides B::F(int32)
    …
  }
}
.class public … D3 extends B {
  .method public newslot virtual final instance void F() … {
    .override B::F      // overrides B::F()
    …
  }
}
```

end example]

34.7.4.2 Sealed function modifier

A ref class function explicitly declared `sealed` shall be emitted as a method marked `final`. [*Example*:

```
public ref struct R {
   virtual void F() sealed { … }
};
```

```
.class … R … {
  .method … final instance void F() … { … }
}
```

end example]

34.7.4.3 Abstract function modifier

A ref class function explicitly declared `abstract` shall be emitted as a method marked `abstract`. [*Example*:

```
public ref struct R {
   virtual void F1() = 0;
   virtual void F2() abstract;
   virtual void F3() abstract = 0;
};
```

```
.class … abstract … R … {
  .method … abstract … void F1() … { … }
  .method … abstract … void F2() … { … }
  .method … abstract … void F3() … { … }
}
```

end example]

All instance functions in an interface class shall be emitted as methods marked `abstract`.

34.7.4.4 The newslot attribute

The `new` function modifier corresponds exactly to the CLI's predefined attribute `newslot`. [*Note:* According to the CLI Standard, Partition II:

> "A virtual method is introduced in the inheritance hierarchy by defining a virtual method. The versioning semantics differ depending on whether or not the definition is marked as `newslot`:
>
> If the definition is marked `newslot` then the definition always creates a new virtual method, even if a base class provides a matching virtual method. Any reference to the virtual method created before the new virtual function was defined will continue to refer to the original definition.
>
> If the definition is not marked `newslot` then the definition creates a new virtual method only if there is no virtual method of the same name and signature inherited from a base class. If the

inheritance hierarchy changes so that the definition matches an inherited virtual function, the definition will be treated as a new implementation of that inherited function."

end note]

Functions shall be marked `newslot` in the following cases only:

- The function is a member of an interface.
- The function is a virtual function in a ref class or value class and that function's name is not seen by lookup in any of the base classes. [*Note*: Lookup ignores interfaces, so if the name is specified only in an interface, the function is still marked as `newslot`. *end note*]
- The function is a virtual function declared using `new`.

34.7.4.5 Special attributes

The attributes `InAttribute` and `OutAttribute` (both in namespace `System::Runtime::InteropServices`) can be applied to function parameters. The compiler is required to not preserve these types in metadata as custom attributes. Instead, the compiler shall emit them directly in the file format. (Consumers of such metadata are required to retrieve this data from the file format and return it as if it were a custom attribute.) [*Example*:

```
public ref struct C {
   void F(int* p1, [In] int* p2, [Out] int* p3, [In, Out] int* p4) { … }
};
```

```
.class public … C … {
  .method public instance void F(int32* p1, [in] int32* p2,
      [out] int32* p3, [in][out] int32* p4) … { … }
}
```

end example]

A method definition can be marked with a variety of implementation attributes. Some of these can be specified via the attribute `MethodImplAttribute` (in namespace `System::Runtime::CompilerServices`), which takes as an argument, one or a combination of enumerators from the type `MethodImplOptions` (also in the same namespace). The compiler is required to not preserve this type in metadata as a custom attribute. Instead, the compiler shall emit it directly in the file format. (Consumers of such metadata are required to retrieve this data from the file format and return it as if it were a custom attribute.) [*Example*:

```
public ref struct C {
   [MethodImpl(MethodImplOptions::NoInlining)] void F1() { … }
   [MethodImpl(MethodImplOptions::Synchronized |
    MethodImplOptions::NoInlining)] void F2() { … }
};
```

```
.class public … C … {
  .method public instance void  F1() … noinlining { … }
  .method public instance void  F2() … synchronized
     noinlining { … }
}
```

end example]

34.7.5 Properties

A property shall be emitted as a `.property` directive plus one `.method` directive for each accessor. No other methods shall be emitted. If the property has a get accessor function, the `.property` directive shall contain a `.get` directive. If the property has a set accessor function, the `.property` directive shall contain a `.set` directive. The method definitions shall be marked `specialname`. A property itself shall not be marked `rtspecialname` or `specialname`.

The definition of an instance property shall be marked `instance`. Any `.set` and `.get` directives that property contains shall also be marked `instance`, as shall the corresponding method definitions. For a static property, only the method definition shall be marked `static`.

For a scalar or named indexed property `P`, the name of the method emitted for a get accessor function shall be `get_P`, while that for a set accessor function shall be `set_P`. For a default-indexed property declared in a type not having the attribute `DefaultMemberAttribute`, the metadata emitted shall be as if that property were a named indexed property called `Item`. For a default-indexed property declared in a type having the attribute `DefaultMemberAttribute`, the metadata emitted shall be as if that property were a named indexed property having the name specified by that attribute.

The accessibility of a property shall be reflected in the definitions of its `.method`s. (See §34.7.2.) [*Note*: The get and set accessor functions of a property can have different accessibilities. *end note*]

[*Example*:

```
public value class Point {
   static int pointCount = 0;
   int x;
   int y;
public:
   property int X {
      int get() { return x; }
      void set(int val) { x = val; }
   }

   …
   static property int PointCount {
      int get() { return pointCount; }
   }
};
```

```
.class public … Point … {
  …
  .property instance int32 X() {
    .set instance void Point::set_X(int32)
    .get instance int32 Point::get_X()
  }
  .method public specialname instance int32 get_X() … { … }
  .method public specialname instance void set_X(int32 val) … { … }
  .property int32 PointCount() {
    .get int32 Point::get_PointCount()
  }
  .method public specialname static int32 get_PointCount() … { … }
}
```

end example] [*Example*:

```
public ref class IntVector {
   int length;
   array<int>^ values;
public:
   property int default[int] {
      int  get(int index)              { return values[index]; }
      void set(int index, int value)   { values[index] = value; }
   }
}
```

```
.class public … IntVector … {
  .field private int32 length
  .field private int32[] values
```

```
  .property instance int32 Item(int32) {
    .get instance int32 IntVector::get_Item(int32)
    .set instance void  IntVector::set_Item(int32, int32)
  }

  .method public … int32 get_Item(int32 index) … { … }
  .method public … void set_Item(int32 index, int32 value) … { … }
}
```

end example]

If a property is declared `virtual`, the accessor methods it has shall be marked `newslot virtual`. If a property is not declared `virtual`, but either of the two of its accessors, or its only accessor is, then the accessor emitted shall be marked `newslot virtual`.

If a property is declared `sealed`, the accessor methods it has shall be marked `newslot virtual final`. If a property is not declared `sealed`, but either of the two of its accessors, or its only accessor is, then the accessor emitted shall be marked `newslot virtual final`.

If a property is declared `abstract`, the accessor methods it has shall be marked `newslot abstract virtual`. If a property is not declared `abstract`, but either of the two of its accessors, or its only accessor is, then the accessor emitted shall be marked `newslot abstract virtual`.

In the case of a trivial scalar property, the private backing storage field allocated shall have a name in the implementer's namespace, and be an instance or static field, as appropriate. [*Example*:

```
public ref struct C {
   property int P;
};
```

```
.class public … C … {
  .field private int32 '<backing_store>P'

  .property instance int32 P() {
    .set instance void C2::set_P(int32)
    .get instance int32 C2::get_P()
  }

  .method … int32 get_P() … {
    .maxstack  1
    .locals (int32 V_0)
    ldarg.0
    ldfld      int32 C2::'<backing_store>P'
    stloc.0
    ldloc.0
    ret
  }

  .method … void set_P(int32 __set_formal) … {
    .maxstack  2
    ldarg.0
    ldarg.1
    stfld      int32 C2::'<backing_store>P'
    ret
  }
}
```

end example]

The accessor methods of a property can be marked with a variety of implementation attributes. For more information see §34.7.4.

34.7.6 Events

An event is implemented via an `.event` directive. That directive shall refer to one add and one remove accessor function by using an `.addon` and a `.removeon` directive, respectively. For an event having a raise accessor function, that function shall be referred to in the `.event` directive using a `.fire` directive. The name of the add, remove, and raise accessor functions shall be `add_`*xx*, `remove_`*xx*, and `raise_`*xx*, respectively, where *xx* is the declared name of the event. All accessor functions shall be marked

`specialname`. If the add or remove accessor functions have the attribute `MethodImpl(MethodImplOptions::Synchronized)`, the resulting methods shall be marked `synchronized` (see §34.7.4). [*Example*:

```
public delegate void EvtHandler(Object^ sender, EventArgs^ e);

public ref class Button {
   EvtHandler^ action;
public:
   event EvtHandler^ Click {
      [MethodImpl(MethodImplOptions::Synchronized)]
      void add(EvtHandler^ d) {}
      [MethodImpl(MethodImplOptions::Synchronized)]
      void remove(EvtHandler^ d) { … }
      void raise(Object^ sender, EventArgs^ e) { … }
   }
};
```

```
.class public … Button … {
  .field private class EvtHandler action

  .event specialname EvtHandler Click {
    .addon instance void Button::add_Click(class EvtHandler)
    .removeon instance void Button::remove_Click(class EvtHandler)
    .fire instance void Button::raise_Click(object,
       class [mscorlib]System.EventArgs)
  }

  .method public specialname instance void add_Click(class EvtHandler d)
     … synchronized { … }

  .method public specialname instance void remove_Click(class
     EvtHandler d) … synchronized { … }

  .method public specialname instance void raise_Click(object sender,
     class [mscorlib]System.EventArgs e) … { … }
}
```

end example]

A trivial event is handled in much the same way as a non-trivial one, except that for a trivial event, storage shall be allocated for a field to hold the delegate, and add, remove, and raise accessor functions shall be generated to add and remove functions from the delegate field, and raise the event, respectively. The generated add and remove accessor functions shall have the same access specifier as their parent event. The generated raise accessor function shall be marked `family`.

The generated add accessor function shall combine the delegate argument passed to it with the delegate field. The generated remove accessor function shall remove the delegate argument passed to it from the delegate field. The generated raise accessor function shall call the delegate field's `Invoke` method, passing it the argument list the raise accessor function was given; that accessor function shall return the value returned by that call to `Invoke`. In order to be thread-safe, the generated add and remove accessor functions shall be marked `synchronized`. The generated raise access function shall not be so marked. [*Example*:

```
public delegate int D(int);

public ref struct X {
   event D^ Ev;
};
```

```
.class public … X … {
  .field private class D '<Ev>'

  .event specialname D Ev {
    .addon instance void X::add_Ev(class D)
    .removeon instance void X::remove_Ev(class D)
    .fire instance int32 X::raise_Ev(int32)
  }
```

```
  .method public specialname instance void add_Ev(class D '<value>')
    … synchronized {
      …
      ldfld class D X::'<Ev>'
      …
      call class [mscorlib]System.Delegate
        [mscorlib]System.Delegate::Combine(class
        [mscorlib]System.Delegate, class [mscorlib]System.Delegate)
          …
      stfld class D X::'<Ev>'
      …
  }
  .method public specialname instance void remove_Ev(class D '<value>')
    … synchronized {
      …
      ldfld class D X::'<Ev>'
      …
      call class [mscorlib]System.Delegate
        [mscorlib]System.Delegate::Remove(class [mscorlib]System.Delegate,
        class [mscorlib]System.Delegate)
          …
      stfld class D X::'<Ev>'
      …
  }
  .method family specialname instance int32 raise_Ev(int32 value0) … {
    …
    ldfld class D X::'<Ev>'
    …
    callvirt instance int32 D::Invoke(int32)
    …
    ret
  }
}
```

end example]

34.7.7 Static operators

When an implementation emits metadata for a CLS-compliant operator, it shall translate the C++ operator function identifier to its respective CLS-compliant name, as shown in Table 19-1: CLS-Compliant Unary Operators and Table 19-2: CLS-Compliant Binary Operators. When an implementation imports functions from metadata, it shall rewrite that function's CLS-compliant name as its corresponding C++ operator function identifier, as indicated by these tables..

If an operator function does not match the criteria for a CLS-compliant operator (§19.7.5.1), the operator is C++-dependent. Table 19-4: C++-Dependent Unary Operators and Table 19-5: C++-Dependent Binary Operators identify these functions.

When an implementation imports C++-dependent functions (Table 19-4: C++-Dependent Unary Operators and Table 19-5: C++-Dependent Binary Operators) from metadata, these functions shall be treated using their corresponding C++ identifiers. If such a function does not make sense as an operator function (for example, it takes three arguments), the function name shall not be changed to the internal operator function name, and the function shall be callable by the name it has in the metadata.

All static operator functions shall be marked `static` and `specialname`.

[*Example*:

```
public ref class IntVector {
   …
public:
   static IntVector^ operator+(IntVector^ iv, int i);
   static IntVector^ operator+(int i, IntVector^ iv);
   static IntVector^ operator+(IntVector^ iv1, IntVector^ iv2);
   static IntVector^ operator-(IntVector^ iv);
   static IntVector^ operator++(IntVector^ iv);
   …
};
```

```
.class public … IntVector … {
  .method public specialname static class IntVector op_Addition(
    class IntVector iv, int32 val) … { … }
  .method public specialname static class IntVector op_Addition(
    int32 val, class IntVector iv) … { … }
  .method public specialname static class IntVector op_Addition(
    class IntVector iv1, class IntVector iv2) … { … }
  .method public specialname static class IntVector op_UnaryNegation(
    class IntVector iv) … { … }
  .method public specialname static class IntVector op_Increment(
    class IntVector iv) … { … }
}
```

end example]

34.7.8 Non-static operators

The metadata for non-static operators implemented as member functions is just like that for static operators, except that in the former case, the function is implemented as an instance method instead of a static one.

All non-static operator functions shall be marked `specialname`.

As with Standard C++, instance versions of `operator++` and `operator--` have to be implemented separately for prefix and postfix notation. [*Example*:

```
public ref class IntVector {
   …
public:
   IntVector^ operator+(int val);
   static IntVector^ operator+(int val, IntVector^ iv);
   IntVector^ operator+(IntVector^ iv2);
   IntVector^ operator-();
   IntVector^ operator++();
   IntVector^ operator++(int);
   …
};
```

```
.class public … IntVector … {
  .method public specialname class IntVector op_Addition(int32 val)
    … { … }
  .method public specialname static class IntVector op_Addition(
    int32 val, class IntVector iv) … { … }
  .method public specialname class IntVector op_Addition(
    class IntVector iv2) … { … }
  .method public specialname class IntVector op_UnaryNegation() … { … }
  .method public specialname class IntVector op_Increment() … { … }
  .method public specialname class IntVector op_Increment(int32) … { … }
}
```

The function `operator+(int, Intvector^)` cannot be implemented as an instance method as its first parameter is not of the parent class type or a handle to that type. *end example*]

In the case of operators implemented as global functions, they shall be marked `assembly`, and their names shall be the exact spelling of their source language token; `'+'` for `operator+`, `'-'` for `operator-`, `'++'` for `operator++`, and so on. As with Standard C++, instance versions of `operator++` and `operator--` have to be implemented separately for prefix and postfix notation. [*Example*:

```
public ref class IntVector {
   …
};

IntVector^ operator+(IntVector^ iv, int val);
IntVector^ operator+(int val, IntVector^ iv);
IntVector^ operator+(IntVector^ iv1, IntVector^ iv2);
IntVector^ operator-(IntVector^ iv);
IntVector^ operator++(IntVector^ iv);
IntVector^ operator++(IntVector^ iv, int);
```

```
.class public … IntVector … {
  …
}

.class public abstract … '…' {
  .method assembly specialname static class IntVector '+'(
    class IntVector iv, int32 val) … { … }

  .method assembly specialname static class IntVector '+'(
    int32 val, class IntVector iv) … { … }

  .method assembly specialname static class IntVector '+'(
    class IntVector iv1, class IntVector iv2) … { … }

  .method assembly specialname static class IntVector '-'(
    class IntVector iv) … { … }

  .method assembly specialname static class IntVector '++'(
    class IntVector iv) … { … }

  .method assembly specialname static class IntVector '++'(
    class IntVector iv, int32) … { … }
}
```

end example]

34.7.9 Instance constructors

An instance constructor of a ref class shall be emitted as an instance method, called `.ctor`, of its class. The accessibility of the constructor shall be reflected in its definition (see §34.7.2). The method shall be marked `specialname`, `rtspecialname`, `instance`, `cil`, and `managed`, and shall have a `void` return type and corresponding parameter list. [*Example*:

```
public ref class C {
   int v;
   C() { … }
public:
   C(int i) : v(i) { … }
};
```

```
.class public … C … {
  .method private specialname rtspecialname instance void .ctor() … {
    .maxstack …
    ldarg.0
    call       instance void [mscorlib]System.Object::.ctor()
    …
    ret
  }
```

```
  .method public specialname rtspecialname instance void .ctor(int32 i) …
{
    .maxstack …
    ldarg.0
    call       instance void [mscorlib]System.Object::.ctor()
    ldarg.0
    ldarg.1
    stfld      int32 C::v
    …
    ret
  }
}
```

end example]

An instance constructor can be marked with a variety of implementation attributes. For more information see §34.7.4.

34.7.10 Static constructors

A static constructor of a ref or value class shall be emitted as a `private` static method, called `.cctor`, of its class. The method shall be marked `specialname`, `rtspecialname`, `static`, `cil`, and `managed`, and shall have a `void` return type and no arguments. The class itself shall be marked `beforefieldinit`. [*Example*:

```
public ref class B {
   static B() { … }
public:
   …
};
```

```
.class public beforefieldinit … B … {
  .method private specialname rtspecialname static void .cctor()
      cil managed { … }
}
```

end example]

A static constructor can be marked with a variety of implementation attributes. For more information see §34.7.4.

34.7.11 Literal fields

A literal field shall be implemented as a public static literal field with the specified initial value. [*Example*:

```
public ref struct X {
   literal int Count = 100;
   literal String^ Greeting = "Hello";
};
```

```
.class public … X … {
  .field public static literal int32 Count = int32(0x00000064)
  .field public static literal string Greeting = "Hello"
}
```

end example]

For information about metadata generation for data members in general, see §34.7.3.

34.7.12 Initonly fields

An initonly field shall be implemented as an instance or static initonly field, as appropriate. The accessibility of the field shall be reflected in its definition. The initialization code placed in the static constructor for each explicitly initialized static initonly field shall cause those fields to be initialized in their declaration lexical order. [*Example*:

```
public ref class X {
   initonly static int V1 = 5, V2 = V1;
   initonly static int V3 = V2 + 1;
   initonly static int V4;
public:
   initonly int V5;
   static X() { V4 = V1 + V3; }
   X(int i) { V5 = i; }
};
```

```
.class public … X … {
  .field private static initonly int32 V1
  .field private static initonly int32 V2
  .field private static initonly int32 V3
  .field private static initonly int32 V4
  .field public initonly int32 V5

  .method private specialname rtspecialname static void .cctor() … {
    .maxstack  2
    ldc.i4.5
    stsfld     int32 X::V1

    ldsfld     int32 X::V1
    stsfld     int32 X::V2

    ldsfld     int32 X::V2
    ldc.i4.1
    add
    stsfld     int32 X::V3

    ldsfld     int32 X::V1
    ldsfld     int32 X::V3
    add
    stsfld     int32 X::V4
    ret
  }
}
```

In the static constructor, `V1`, `V2`, and `V3` shall be initialized in that order, all before the assignment to `V4`. *end example*]

For information about metadata generation for data members in general, see §34.7.3.

34.7.13 Destructors and finalizers

34.7.13.1 CLI dispose pattern

C++/CLI implements the destructor and finalizer semantics in ref classes by using the CLI dispose pattern. This pattern makes use of three functions upon which all languages targeting the CLI agree. These functions are

```
void Dispose();
void Dispose(bool);
void Finalize();
```

and their definitions are generated by the compiler, as required. Two other C++/CLI-specific private helper functions are also generated, and used by `Dispose(bool)`; they are:

```
void __identifier("~T")()
void __identifier("!T")()
```

where `T` is the parent class name.

Many languages have constructs that support this dispose pattern directly. Since C++/CLI fully supports this dispose pattern, any CLI class type authored in C++/CLI can be used by other languages, and any CLI class type authored in other languages and having this dispose pattern, supports C++ destructor cleanup semantics when used in C++/CLI code.

The CLI dispose pattern requires the following:

- A function `Dispose()` that implements `System::IDisposable::Dispose()`.
- A function `Finalize()` that overrides `System::Object::Finalize()`.
- A function `Dispose(bool)`, which is a member of a class that has a `Dispose()` function that implements `System::IDisposable::Dispose()`, or is a member of a class that has a `Finalize()` function that overrides `System::Object::Finalize()`, or the `Dispose(bool)` function itself overrides a `Dispose(bool)` function in a base class that does have such a `Dispose()` or `Finalize()` function.

A C++/CLI program that contains a definition for a function having any of these signatures is ill-formed. [*Note*: It would be helpful to the programmer if the diagnostic issued in such cases encouraged the programmer to define a destructor and/or finalizer instead. *end note*] Function definitions having these signatures can exist, however.

If a function definition having any of these signatures fulfills the corresponding requirement above, it shall be used to implement the CLI dispose pattern, and a C++/CLI program that calls such a function is ill-formed. [*Note*: It would be helpful to the programmer if the diagnostic issued in such cases encouraged the programmer to call the destructor instead. *end note*] If a function definition having any of these signatures does not fulfill the corresponding requirement above, it shall not be used to implement the CLI dispose pattern, and a C++/CLI program is permitted to call that function directly.

The `System::IDisposable` interface is used by the CLI dispose pattern as an entry point for destruction. However, because C++/CLI provides direct support for cleanup via destructors and finalizers, the `System::IDisposable` interface need never be used directly. A C++/CLI program shall not use this interface.

[*Example*:

```
public ref class B {
protected:
   !B() {}
public:
   ~B() {}
};

public ref class D : B {
protected:
   !D() {}
public:
   ~D() {}
};
```

```
.class … B … implements [mscorlib]System.IDisposable {
  .method … void  '!B'() … { … }
  .method … void Dispose(bool marshal( unsigned int8) A_1) … {
      ldarg.1
      brfalse.s  IL_000b
      ldarg.0
      call       instance void B::'~B'()
      br.s       IL_001b
    IL_000b:
      nop

    .try {
      ldarg.0
      call       instance void B::'!B'()
      leave.s    IL_001b
    }

    finally {
      ldarg.0
      call instance void [mscorlib]System.Object::Finalize()
      endfinally
    }
    IL_001b:
      ret
  }
```

```
  .method … void Dispose() … {
      ldarg.0
      ldc.i4.1
      callvirt   instance void B::Dispose(bool)
      ldarg.0
      call       void [mscorlib]System.GC::SuppressFinalize(object)
      ret
  }
  .method … void Finalize() … {
      ldarg.0
      ldc.i4.0
      callvirt   instance void B::Dispose(bool)
      ret
  }
  .method … void  '~B'() … { … }
}
.class … D extends B {
  .method … void  '!D'() … { … }
  .method … void Dispose(bool marshal( unsigned int8) A_1) … {
      ldarg.1
      brfalse.s  IL_0015

    .try {
      ldarg.0
      call       instance void D::'~D'()
      leave.s    IL_0013
    }
    finally {
      ldarg.0
      ldc.i4.1
      call       instance void B::Dispose(bool)
      endfinally
    }
  IL_0013:
      br.s       IL_0026
  IL_0015:
      nop

    .try {
      ldarg.0
      call       instance void D::'!D'()
      leave.s    IL_0026
    }
    finally {
      ldarg.0
      ldc.i4.0
      call       instance void B::Dispose(bool)
      endfinally
    }
  IL_0026:
      ret
  }
  .method … void  '~D'() … { … }
}
```

end example]

34.7.13.2 Destructors

A ref class with a user-defined or compiler-generated destructor shall be marked as implementing `System::IDisposable`.

Destruction of an instance of a ref class shall always begin by dynamically casting that object to `System::IDisposable`. If that cast succeeds, the `Dispose()` function shall be called through the result

of the cast. If that cast fails, the destructor does nothing. [*Note*: As a result, a destructor can be called on an instance of any ref class, value class, or interface class. *end note*]

The compiler shall not generate code to call a destructor except through the `System::IDisposable::Dispose` function.

Although a value class cannot have a destructor, if a value class indirectly implements `System::IDisposable` (as the result of another interface's implementing `System::IDisposable`), the compiler shall emit a corresponding `Dispose()` function that implements the interface; however, that `Dispose()` function shall do nothing.

For an interface class declaring a destructor, no method shall be emitted for that destructor; however, the interface shall be marked as implementing `System::IDisposable`.

34.7.13.3 Finalizers

A finalizer for a class shall be generated if and only if the user writes a finalizer for that class.

Calls to a finalizer in any ref class `T` result in direct calls to the `__identifier("!T")` function (§34.7.13.9).

34.7.13.4 Functions generated to support the dispose pattern

The CLI dispose pattern uses three primary functions: `Dispose()`, `Finalize()`, and `Dispose(bool)`. Two secondary functions, `__identifier("~T")()` and `__identifier("!T")()`, are called by `Dispose(bool)`. The definitions of all five functions are generated by the compiler, as specified below.

34.7.13.5 The Dispose() function

This member function is the starting point for cleanup done via destruction.

This function shall only be emitted for any ref class `T` in the following scenarios:

- The `Dispose(bool)` function is being introduced by class `T` (Cases #2 and #3 below), or
- If Case #1 was used and no base class that used Case #1 has already introduced a `public virtual Dispose()` that implements `System::IDisposable`.

This function shall not be emitted

- If the dispose pattern already exists, and
- A `Dispose()` that is part of the dispose pattern also exists, and
- The class explicitly implements `System::IDisposable`.

This function shall be emitted as if it were written in C++/CLI, inside the definition of `T`, as follows:

```
public:
   virtual void Dispose() sealed {
      this->Dispose(true);
      System::GC::SuppressFinalize(this);
   }
```

The parent class of any `Dispose()` function emitted by the compiler, shall be marked as implementing `System::IDisposable`.

If a base class of `T` has a `Dispose()` method that does not implement `System::IDisposable`, that base class function shall be hidden by the one emitted for `T`. The `Dispose()` function shall be marked `newslot` in metadata unless the function can override a base class's implementation of `Dispose()` that implements `System::IDisposable`.

34.7.13.6 The Finalize() function

This function is the starting point for cleanup done via finalization.

This function shall only be emitted for any ref class `T` if the following criteria are met:

- The compiler will generate an `__identifier("!T")` function for class `T`, and
- Class `T` is introducing the dispose pattern (Cases #2 and #3 below), or if class `T` is extending the dispose pattern (Case #1 below), no base class with the dispose pattern has already introduced a `Finalize()` function.

This function shall be emitted as if it were written in C++/CLI, inside the definition of `T`, as follows:

```
protected:
   virtual void Finalize() override {
      this->Dispose(false);
   }
```

The `Finalize()` function shall never be marked `newslot` in metadata.

34.7.13.7 The Dispose(bool) function

For any ref class `T`, this function is generated if and only if either or both of the functions `__identifier("~T")()` and `__identifier("!T")()` are generated for this class or the compiler needs to generate a non-trivial destructor to clean up members of that class.

This function has three possible forms, as shown in Case #1, Case #2, and Case #3, below. (In each Case, the base class of `T` is assumed to be `Base`. It is also assumed that class `T` has both a destructor and a finalizer. If one or the other of these functions is omitted, the corresponding call to `__identifier("~T")` or `__identifier("~T")` shall be omitted.) The decision tree following these Cases shows how each Case is chosen.

Case #1: Extending the dispose pattern, existing Dispose(bool) that is part of the dispose pattern

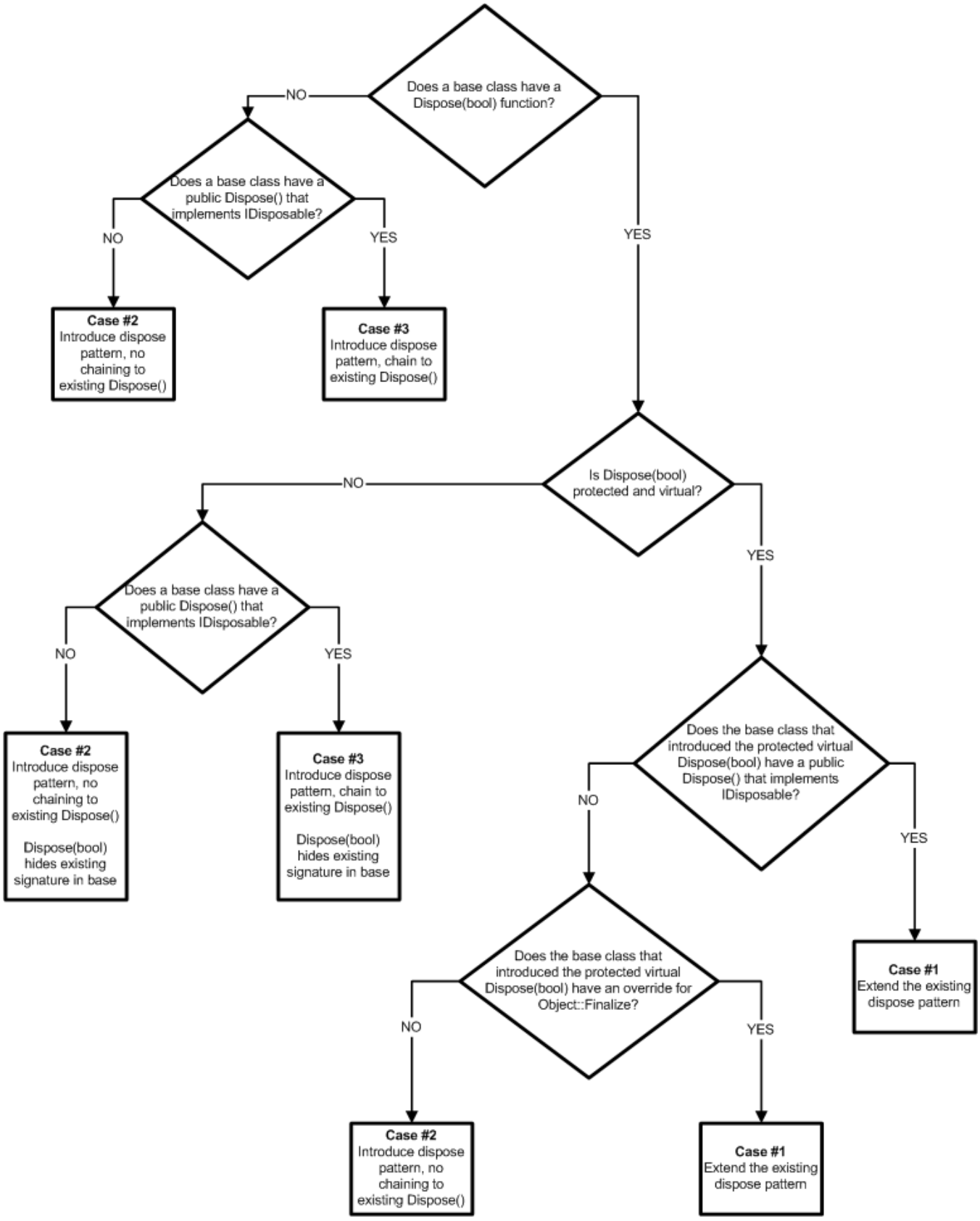
```
protected:
   virtual void Dispose(bool calledFromDispose) override {
      if (calledFromDispose) {
         try {
            this->__identifier("~T")();
         } finally {
            try {
               this->Base::Dispose(true);
            } finally {
               // member cleanup goes here
            }
         }
      } else {
         try {
            this->__identifier("!T")();
         } finally {
            this->Base::Dispose(false);
         }
      }
   }
```

Case #2: Introducing dispose pattern, no public Dispose() that implements System::IDisposable

```
protected:
   virtual void Dispose(bool calledFromDispose) {
      if (calledFromDispose) {
         this->__identifier("~T")();
      } else {
         try {
            try {
               this->__identifier("!T")();
            } finally {
               // member cleanup goes here
            }
         } finally {
            this->Base::Finalize();
         }
      }
   }
```

Case #3: Introducing dispose pattern, existing callable Dispose()

```
protected:
   virtual void Dispose(bool calledFromDispose) {
      if (calledFromDispose) {
         try {
            this->__identifier("~T")();
         } finally {
            try {
               this->Base::Dispose();
            } finally {
               // member cleanup goes here
            }
         }
      } else {
         try {
            this->__identifier("!T")();
         } finally {
            this->Base::Finalize();
         }
      }
   }
```


Does a base class have a Dispose(bool) function?
NO
YES
Does a base class have a public Dispose() that implements IDisposable?
NO
YES
Case #2
Introduce dispose pattern, no chaining to existing Dispose()
Case #3
Introduce dispose pattern, chain to existing Dispose()
Is Dispose(bool) protected and virtual?
NO
YES
Does a base class have a public Dispose() that implements IDisposable?
NO
YES
Case #2
Introduce dispose pattern, no chaining to existing Dispose()
Dispose(bool) hides existing signature in base
Case #3
Introduce dispose pattern, chain to existing Dispose()
Dispose(bool) hides existing signature in base
Does the base class that introduced the protected virtual Dispose(bool) have a public Dispose() that implements IDisposable?
NO
YES
Case #1
Extend the existing dispose pattern
Does the base class that introduced the protected virtual Dispose(bool) have an override for Object::Finalize?
NO
YES
Case #2
Introduce dispose pattern, no chaining to existing Dispose()
Case #1
Extend the existing dispose pattern

34.7.13.8 The __identifier("~T")() function

This function shall be emitted for any ref class `T`, but only if that class has a user-defined destructor. The body of this function shall correspond exactly to that of the user-defined destructor. The compiler shall not generate calls to functions in the base class in this function.

This function shall be emitted as if it were written in C++/CLI, inside the definition of `T`, as follows:

```
private:
   void __identifier("~T")() {
      // user-defined destructor body goes here
   }
```

34.7.13.9 The __identifier("!T")() function

This function shall be emitted for any ref class `T`, but only if that class has a user-defined finalizer. The body of this function shall correspond exactly to that of the user-defined finalizer. The compiler shall not generate any other code in this function.

This function shall be emitted as if it were written in C++/CLI, inside the definition of `T`, as follows:

```
private:
   void __identifier("!T")() {
      // user-defined finalizer body goes here
   }
```

34.8 Native classes

A native class shall be emitted as a value class (even though a native class is *not* a value class) with the corresponding name and visibility (§34.6.3). It shall be marked with the following:

- The "Marshal string" attributes `ansi` (§34.7.3), and
- The "Type layout" attribute `sequential` (§34.7.3),

however, the corresponding attribute, `StructLayoutAttribute` (and `FieldOffsetAttribute`), from namespace `System::Runtime::InteropServices` cannot be applied to a native class at the source code level.
A nested native class or value class shall be marked `nested`, followed by the appropriate accessibility, and shall be defined inside the type in which it is nested.

Like a value class, a native class shall extend `[mscorlib]System::ValueType`.

The value class used to encode the native class shall contain an explicit `.size` directive whose value is determined by the implementation, as the size needed to represent an instance of that class.

The value class used to encode the tnative class shall have attached to it the `NativeCppClass` (§33.2.1) attribute, from namespace `System::Runtime::CompilerServices`.

The encoding for a native class is not required to have any other characteristics. In particular, it is not required to have a constructor or the members of the class encoded.

[*Example*:

```
public class N1 {
   char c[2];
   int i;
   double d;
public:
   void F() { … }
};
```

```
.class public sequential ansi sealed N1 extends
    [mscorlib]System.ValueType {
  .size 16
  .custom instance void [mscorlib]System.Runtime.CompilerServices.
    NativeCppClassAttribute::.ctor() = ( … )}
```

The size 16 bytes is based on an implementation in which a `char` occupies 1 byte, an `int` occupies 4 bytes, a `double` occupies 8 bytes, a `char` can be aligned on any boundary, an `int` is aligned on a 4-byte boundary, and a `double` is aligned on an 8-byte boundary. (That is, two 1-byte `char`s, two bytes of padding, one 4-byte `int`, and one 8-byte `double`.)

```
namespace MyApp {
public class N2 {
   char c[3];
   double d;
   int i;

public:
   void F(int i) {  }
   class N3 {
      short int s;
   public:
      void F(int i) {  }
   };
};
}
```

```
.class public sequential ansi sealed MyApp.N2 extends
     [mscorlib]System.ValueType {
  .size 24
  .custom instance void [mscorlib]System.Runtime.CompilerServices.
    NativeCppClassAttribute::.ctor() = ( … )
  .class sequential ansi sealed nested public N3 extends
     [mscorlib]System.ValueType {
    .size 2
    .custom instance void [mscorlib]System.Runtime.CompilerServices.
    NativeCppClassAttribute::.ctor() = ( … )
  }
}
```

The size 24 bytes comes from three 1-byte `char`s, five bytes of padding, one 8-byte `double`, one 4-byte `int`, one 2-byte `short`, and two bytes of padding. The size 2 bytes comes from one 2-byte `short`.

```
template<typename T>
public class N4 {
   T m1;
   T m2[2];
public:
   void F(T t, T* pt) {}
};

N4<char> n4a;
N4<int> n4b;
```

```
.class public sequential ansi sealed 'N4<char>' extends
     [mscorlib]System.ValueType {
  .size 3
  .custom instance void [mscorlib]System.Runtime.CompilerServices.
    NativeCppClassAttribute::.ctor() = ( … )
}

.class public sequential ansi sealed 'N4<int>' extends
     [mscorlib]System.ValueType {
  .size 12
  .custom instance void [mscorlib]System.Runtime.CompilerServices.
    NativeCppClassAttribute::.ctor() = ( … )
}
```

The encodings of `n4a` and `n4b` are not shown.

end example]

Metadata for template classes is described in §34.17.

34.9 Ref classes

[*Note*: For implementations providing the `IsBoxed` modifier: Any member function of a ref class, value class, or interface class having a parameter declaration or return type involving a handle to a value type shall have that parameter and/or return type marked with the modifier `IsBoxed` (§33.1.5.1). *end note*]

Any member function of a ref class, value class, or interface class having a ref class type parameter passed by value shall have the corresponding parameter marked with the modifier `IsByValue` (§33.1.5.2).

Any member function of a ref class, value class, or interface class having a `const`-qualified parameter or returning a `const`-qualified type shall have the corresponding parameter and/or return type marked with the modifier `IsConst` (§33.1.5.3), as appropriate. However, parameter qualification at the top level shall not be so marked. [*Example*: A parameter such as `const int* ci` shall be marked, but one such as `const int i` shall not. *end example*]

Any data member of a ref class, value class, or interface class having a `const`-qualified type shall be marked with the modifier `IsConst` (§33.1.5.3).

Any member function of a ref class, value class, or interface class having a parameter that is an interior pointer or pinning pointer shall have the corresponding parameter marked with the modifier `IsExplicitlyDereferenced` (§33.1.5.4).

Any member function of a ref class, value class, or interface class having a parameter that is a reference or tracking reference, or returning a reference or tracking reference shall have the corresponding parameter and/or return type marked with the modifier `IsImplicitlyDereferenced` (§33.1.5.5).

Any data member of a ref class, value class, or interface class that is a reference or tracking reference shall be marked with the modifier `IsImplicitlyDereferenced` (§33.1.5.5).

[*Note*: For implementations providing the `IsLong` modifier: Any member function of a ref class, value class, or interface class having a parameter declaration or return type involving a `long int` or `long double` shall have that parameter and/or return type marked with the modifier `IsLong` (§33.1.5.6).

Any data member of a ref class, value class, or interface class involving a `long int` or `long double` shall have that parameter and/or return type marked with the modifier `IsLong` (§33.1.5.6). *end note*]

Any member function of a ref class, value class, or interface class having a parameter declaration or return type involving a plain `char` shall have that parameter and/or return type marked with the modifier `IsSignUnspecifiedByte` (§33.1.5.7).

Any data member of a ref class, value class, or interface class involving a plain `char` shall be marked with the modifier `IsSignUnspecifiedByte` (§33.1.5.7).

Any member function of a ref class, value class, or interface class returning an instance of a ref class type by value shall be marked with the modifier `IsUdtReturn` (§33.1.5.8).

Any member function of a ref class, value class, or interface class having a `volatile`-qualified parameter or returning a `volatile`-qualified type shall have the corresponding parameter and/or return type marked with the modifier `IsVolatile` (§33.1.5.9), as appropriate. However, parameter qualification at the top level shall not be so marked. [*Example*: A parameter such as `volatile int* vi` shall be marked, but one such as `volatile int v` shall not. *end example*]

Any data member of a ref class, value class, or interface class having a `volatile`-qualified type shall be marked with the modifier `IsVolatile` (§33.1.5.9).

For more information, see §34.7.1.

34.10 Value classes

For more information, see §34.7.1 and §34.9.

34.11 CLI arrays

CLI arrays are encoded in metadata according to the CLI standard, primarily in Partitions I, II, and III. [*Note*: A CLI array type shall be defined by specifying the element type of the CLI array, the rank of the CLI array, and the upper and lower bounds of each dimension of the CLI array.

CLI array elements shall be laid out within the CLI array object in row-major order. The actual storage allocated for each CLI array element can include platform-specific padding.

The VES shall provide two constructors for arrays:

- The first takes a sequence of integers giving the number of elements in each dimension (a lower bound of zero is assumed).
- The second takes twice as many arguments. These arguments occur in pairs—one pair per dimension—with the first argument of each pair specifying the lower bound for that dimension, and the second argument specifying the total number of elements in that dimension.

In addition to array constructors, the VES provides the instance methods `Get`, `Set`, and `Address` to access specific elements and compute their addresses. These methods take a number for each dimension, to specify the target element. In addition, `Set` takes an additional final argument specifying the value to be stored into the target element. *end note*]

[*Example*:

```
ref class R {
   array<int>^ m1;
   array<array<String^>^, 2>^ m2;
public:
   array<String^, 2>^ F(array<R^, 3>^ ary) { … }
};
```

```
.class … R … {
  .field private int32[] m1
  .field private string[][0...,0...] m2
  .method public instance string[0...,0...]
    F(class R[0...,0...,0...] ary) … { … }
}
```

```
array<int>^ array1D = gcnew array<int>(10);
array<int, 3>^ array3D = gcnew array<int, 3>(10, 20, 30);
pin_ptr<int> pp1;
```

```
.method … {
  .locals ([0] int32[0...,0...,0...] V_0,
           [1] int32[] V_1)
           [2] int32& pinned modopt([mscorlib]
            System.Runtime.CompilerServices.IsExplicitlyDereferenced)
            V_2)

  ldnull
  stloc.1
  ldnull
  stloc.0

  ldc.i4.s   10
  newarr     [mscorlib]System.Int32
  stloc.1
  ldloc.1
  ldc.i4.5
  ldc.i4.s   10
  stelem.i4

  ldc.i4.s   10
  ldc.i4.s   20
  ldc.i4.s   30
  newobj     instance void int32[0...,0...,0...]::.ctor(int32,
                  int32, int32)
  stloc.0
```

```
array1D[5] = 10;
array3D[1,2,3] = array3D[4,5,6];
```

```
  ldloc.0
  ldc.i4.1
  ldc.i4.2
  ldc.i4.3

  ldloc.0
  ldc.i4.4
  ldc.i4.5
  ldc.i4.6

  call       instance int32 int32[0...,0...,0...]::Get(int32,
                  int32, int32)
  call       instance void int32[0...,0...,0...]::Set(int32,
                  int32, int32, int32)
```

```
pp1 = &array1D[8];
pp1 = &array3D[7,6,5];
```

```
  stloc.0
  ldloc.1
  ldc.i4.8
  ldelema    [mscorlib]System.Int32
  stloc.2

  ldloc.0
  ldc.i4.7
  ldc.i4.6
  ldc.i4.5
  call       instance int32& int32[0...,0...,0...]::Address(int32,
                  int32, int32)
```

end example]

34.12 Interfaces

An interface class shall be emitted as a class with the corresponding name and visibility. It shall be marked `interface`. As an interface class is a class, see §34.7 and its subordinate subclauses, and §34.9 for metadata details pertaining to classes and their members.

All interface class member functions shall be emitted as `.method`s marked as `newslot`, `abstract`, and `virtual`. [*Example*:

```
public interface struct I {
   void F();
   property int P {
      int get();
      void set(int value);
   }
};
```

```
.class interface public abstract … I {
  .method public newslot abstract virtual instance void F() … { … }

  .property instance int32 P() {
    .get instance int32 I::get_P()
    .set instance void I::set_P(int32)
  }

  .method public newslot … abstract virtual … int32 get_P() … { … }

  .method public newslot … abstract virtual … void set_P(int32 value)
      … { … }
}
```

end example]

[*Example*:

```
public interface struct I1 {
   void F();
};

public interface struct I2 : I1 {
   void G();
   void K();
};

public ref struct B {
   virtual void K() { … }
};

public ref struct D : B, I2 {
   virtual void F() { … }           // implements I1::F
   virtual void H() = I2::G { … }   // implements I2::G
   virtual void G() new { … }       // a new G
                                    // I2::K implemented by B::K
};

public ref struct E abstract : I1 {
   virtual void F() abstract;
};
```

```
.class interface public abstract … I1 {
  .method public newslot abstract virtual instance void F() … { … }
}

.class interface public abstract … I2 implements I1 {
  .method public newslot abstract virtual instance void G() … { … }
  .method public newslot abstract virtual instance void K() … { … }
}

.class public … B … {
  .method public newslot virtual instance void K() … { … }
}

.class public … D extends B implements I2 {
  .method public virtual instance void  F() … { … }
  .method public newslot virtual final instance void H() … {
    .override I2::G
    …
  }
  .method public newslot virtual instance void G() … { … }
}

.class public abstract … E … implements I1 {
  .method public abstract virtual instance void F() … { … }
}
```

end example]

34.13 Enums

Both native and CLI enums shall be implemented as sealed classes that derive from `System::Enum`. The visibility of the enum type shall be reflected in its class's definition. Each enum type's class shall contain a public instance field called `value__` whose type shall be that of the enum's underlying type, which shall be a CLS-compliant integer type. That field shall be marked `rtspecialname` and `specialname`. (For information specific to fields, see §34.7.3.)

Each enumerator in a CLI enum shall have a corresponding public static literal field of the same name, whose type is that of the parent enum type, and whose value is as defined in the *enum-specifier*. [*Note* Enumerators in native enums have no such corresponding fields. As a result, to share their values across separate compilations, a header must be used. *end note*]

[*Example*:

```
public enum Suit : short { Hearts = 1, Spades, Clubs, Diamonds};

enum class Direction { North, South = 10, East, West = 20 };
```

```
.class public … sealed Suit extends [mscorlib]System.Enum {
  .field public specialname rtspecialname int16 value__
}

.class private … sealed Direction extends [mscorlib]System.Enum {
  .field public static literal valuetype Direction East = int32(0x0B)
  .field public static literal valuetype Direction North = int32(0x00)
  .field public static literal valuetype Direction South = int32(0x0A)
  .field public static literal valuetype Direction West = int32(0x14)
  .field public specialname rtspecialname int32 value__
}
```

end example]

34.14 Delegates

A delegate shall be implemented as a sealed class that (ultimately) derives from `System::Delegate`. [*Note*: A delegate class need not derive directly from this class, however. A conforming implementation of the CLI is permitted to extend the required type hierarchy by including intermediate types. For example, a conforming implementation of the CLI could provide a type `System::MulticastDelegate`, which, in turn, is derived from `System::Delegate`. As such, a conforming C++/CLI implementation could derive its delegate classes from `System::MulticastDelegate`, or from a class derived from that class. *end note*]

The visibility of the delegate type shall be reflected in its class's definition.

For each delegate type class, a conforming implementation shall provide a constructor, a method called `Invoke`, and the methods `BeginInvoke` and `EndInvoke` (used for asynchronous processing), as defined by the CLI standard.

[*Example*:

```
public delegate Object^ D(int* pi, array<int>^ a);

.class public … sealed D extends [mscorlib]System.Delegate {
  .method public specialname rtspecialname instance void
      .ctor(object A_1, native int A_2) runtime managed forwardref {}

  .method public newslot virtual instance class
      [mscorlib]System.IAsyncResult BeginInvoke(int32* pi, int32[] a,
         class [mscorlib]System.AsyncCallback callback, object obj)
         runtime managed forwardref {}

  .method public newslot virtual instance object
      EndInvoke(class [mscorlib]System.IAsyncResult result)
         runtime managed forwardref {}

  .method public newslot virtual instance object Invoke(int32* pi,
      int32[] a) runtime managed forwardref {}
}
```

end example]

In §27.2, it states "Each delegate type shall have two constructors, as follows: …" The library class `System::Delegate` has no constructors defined. Instead, as we can see from the metadata example above, one, and only one, constructor is generated for a delegate, and its implementation attributes are `runtime managed` instead of `cil managed`. This is because the constructor is generated at runtime by the VES. Although the C++/CLI syntax supports delegate constructor calls having either one or two arguments, both forms shall be converted to a call to the one constructor that actually exists in metadata. The C++/CLI constructor taking one argument shall be emitted as a call to the two-argument version with `nullptr` as the first argument.

[*Example*:

```
delegate void D(int i);
```

```
ref struct R {
   static  void M1(int a) { }
           void M2(int b) { }
   virtual void M3(int c) { }
};
int main() {
   R^ r = gcnew R;
   D^ d;
   d =  gcnew D(&R::M1);
   d =  gcnew D(r, &R::M2);
   d += gcnew D(r, &R::M3);
}
```

```
.method … main() … {
  …
  .locals ([0] class D V_0,
           [1] class R V_1)

  ldnull
  stloc.1
  ldnull
  stloc.0
  newobj     instance void R::.ctor()
  stloc.1

  ldnull
  ldftn      void R::M1(int32)
  newobj     instance void D::.ctor(object, native int)
  stloc.0

  ldloc.1
  ldftn      instance void R::M2(int32)
  newobj     instance void D::.ctor(object, native int)
  stloc.0

  ldloc.0
  ldloc.1
  dup
  ldvirtftn  instance void R::M3(int32)
  newobj     instance void D::.ctor(object, native int)

  call       class [mscorlib]System.Delegate
               [mscorlib]System.Delegate::Combine(
               class [mscorlib]System.Delegate,
               class [mscorlib]System.Delegate)
  castclass  D
  stloc.0
  …
}
```

end example]

34.15 Exceptions

`try`, `catch`, and `finally` shall be emitted using one or more `.try` directives. [*Example*:

```
int main() {
   try {
      // ...
   }

   catch (NullReferenceException^ ex1) {
      // ...
   }

   catch (IndexOutOfRangeException^ ex2) {
      // ...
   }
```

```
   finally {
      // ...
   }
}
```

```
.method … main() …
{
  …
  .locals ([0] class [mscorlib]System.IndexOutOfRangeException ex2,
           [1] class [mscorlib]System.NullReferenceException ex1)

  .try
  {
    .try
    {
      …
      leave.s  L8
    }
    catch [mscorlib]System.NullReferenceException
    {
      …
      stloc.1
      leave.s  Le
    }
    catch [mscorlib]System.IndexOutOfRangeException
    {
       …
      stloc.0
      leave.s  La
    }
L8:   br.s     Lc
La:   leave.s  L13
Lc:   br.s     L10
Ie:   leave.s  L13
I10:  leave.s  L13
  }
  finally
  {
      …
      endfinally
  }
L13: …
     …
}
```

end example]

The metadata encoding for *exception-declaration*s that declare non-ref class types, or have the form `...`, is unspecified.

34.16 Attributes

If it is not required to be consumed by the compiler, an attribute on a program element shall be emitted into metadata via a `.custom` directive on that element, or, in some cases, to the immediately preceding element declaration. If a program element has multiple attributes, and multiple attributes are permitted, that element shall have one `.custom` directive for each; their ordering is irrelevant.

A custom attribute is declared using the directive `.custom`, followed by the method declaration for a type constructor (i.e., that method's name shall be `.ctor`), optionally followed by an equals sign (=) and a set of byte values in parentheses. The values of the constructor's arguments, if any, shall be specified in the set of bytes in the format specified by the CLI Standard. If there are no arguments, the equals sign and parenthesized set of bytes shall be omitted. As a constructor is an instance method, its `.custom` directive shall contain the `instance` attribute. [*Example*:

```
[AttributeUsage(AttributeTargets::All, AllowMultiple = true,
   Inherited = true)]
public ref class XAttribute : Attribute {
   String^ name;
public:
   XAttribute(String^ name) : name(name) {}
   property String^ Name { String^ get() { return name;} }
};
```

```
.class public … XAttribute extends [mscorlib]System.Attribute {
  .custom instance void
    [mscorlib]System.AttributeUsageAttribute::.ctor(valuetype
    [mscorlib]System.AttributeTargets) = ( 01 00 FF 7F 00 00 02 00 54
    02 0D 41 6C 6C 6F 77 4D 75 6C 74 69 70 6C 65 01 54 02 09 49 6E 68
    65 72 69 74 65 64 01)
  …
}
```

```
[X("refclass")]
public ref class R {
   [X("field")] int count;
public:
   [X("constructor")] R() {}
};
```

```
.class … R … {
  .custom instance void XAttribute::.ctor(string) = ( 01 00 08 72 65
     66 63 6C 61 73 73 00 00 ) // refclass

  .field private int32 count
  .custom instance void XAttribute::.ctor(string) = ( 01 00 05 66 69 65
6C 64
     00 00 )  // field

  .method public specialname rtspecialname instance void .ctor() cil … {
    .custom instance void XAttribute::.ctor(string) = ( 01 00 0B 63 6F
       6E 73 74 72 75 63 74 6F 72 00 00 ) // constructor
  }
}
```

```
[X("valueclass")]
public value struct V {
   [X("method1"),X("method2")] [returnvalue:X("returnvalue")]
   void Display([X("parameter")] int i) {}
};
```

```
.class … V … {
  .custom instance void XAttribute::.ctor(string) = ( 01 00 0A 76 61
     6C 75 65 63 6C 61 73 73 00 00 )  // valueclass

  .method … void Display(int32 i) … {
    .custom instance void XAttribute::.ctor(string) = ( 01 00 07 6D 65
       74 68 6F 64 32 00 00 )  // method2
    .custom instance void XAttribute::.ctor(string) = ( 01 00 07 6D 65
       74 68 6F 64 31 00 00 )  // method1

    .param [0]
    .custom instance void XAttribute::.ctor(string) = ( 01 00 0B 72 65
       74 75 72 6E 76 61 6C 75 65 00 00 ) // returnvalue

    .param [1]
    .custom instance void XAttribute::.ctor(string) = ( 01 00 09 70 61
       72 61 6D 65 74 65 72 00 00 )  // parameter
  }
}
```

`.param [0]` represents the function's return value, while the actual parameter attributes start with `.param [1]`.

```
[X("interfaceclass")]
public interface class I {
   [X("property")]property int Count {
      [X("getter")]int get();
   }
};
```

```
.class interface … I {
  .custom instance void XAttribute::.ctor(string) = ( 01 00 0E 69 6E
     74 65 72 66 61 63 65 63 6C 61 73 73 00 00 ) // interfaceclass
  .property instance int32 Count() {
    .custom instance void XAttribute::.ctor(string) = ( 01 00 08 70 72
       6F 70 65 72 74 79 00 00 ) // property
    .get instance int32 I::get_Count()
  }
  .method public … get_Count() … {
    .custom instance void XAttribute::.ctor(string) = ( 01 00 06 67 65
      74 74 65 72 00 00 )  // getter
  }
}
```

```
[X("nativeclass")]
public class N {
   [X("field")] int count;
public:
   [X("constructor")] N() { … }
   [X("method")][returnvalue:X("returnvalue")]
   void Display([X("parameter")] int) {}
};
```

```
.class … N … {
  .custom instance void XAttribute::.ctor(string) = ( 01 00 0B 6E 61 74
69 76
    65 63 6C 61 73 73 00 00 ) // nativeclass
}
```

As member information for a native class need not be emitted in metadata, only the `.custom` directive for the class itself need be present. *end example*]

Since attributes can be used to customize metadata, they are often referred to as ***custom attributes***. There are two kinds of custom attributes: ***genuine custom attributes*** and ***pseudo-custom attributes***. Custom attributes and pseudo-custom attributes are treated differently, at the time they are defined, as follows:

- A custom attribute is stored directly into the metadata; the blob which holds its defining data is stored as-is. That blob can be retrieved later.
- A pseudo-custom attribute is recognized because its name is one of a short list. Rather than store its blob directly in metadata, that blob is parsed, and the information it contains is used to set bits and/or fields within metadata tables. The blob is then discarded; it cannot be retrieved later.

Pseudo-custom attributes therefore serve to capture user directives, using the same familiar syntax the compiler provides for genuine custom attributes, but these user directives are then stored into the more space-efficient form of metadata tables. Tables are also faster to check at runtime than are genuine custom attributes.

Many custom attributes are invented by higher layers of software. They are stored and returned by the CLI, without its knowing or caring what they mean. However, all pseudo-custom attributes, plus a collection of genuine custom attributes, are of special interest to compilers and to the CLI. The CLI Standard, Partition II, subclause 21 lists the pseudo-custom attributes and distinguished custom attributes, where distinguished means that the CLI and/or compilers need to pay direct attention to them, and their behavior is affected in some way.

The special processing needed for various pseudo-custom attributes is described elsewhere in this clause. Examples include `DllImportAttribute`, `FieldOffsetAttribute`, `InAttribute`, `MarshalAsAttribute`, `MethodImplAttribute`, `OutAttribute`, and `StructLayoutAttribute`.

A conforming implementation needs to be aware of the attribute `AttributeUsageAttribute` (from namespace `System`).

The parameter array ellipses notation (`...`) involves the generation of a `.custom` directive for the attribute `ParamArrayAttribute`, (in namespace `System`). See §34.6.2.

34.17 Templates

The metadata encoding for template classes and functions is unspecified except that the name of any template class emitted shall not be spelled in a CLS-compliant manner.

34.18 Generics

The name of a generic type shall be that type's name as specified in the C++/CLI source, plus a suffix of the form `n, where n is a decimal integer constant (without leading zeros) representing the arity of that type. The name in metadata of a non-generic type shall not have such a suffix. [*Example*:

```
ref class X { … };
```

```
// metadata type name is X
.class public … X … { … }
```

```
generic<typename T>
public ref class X { … };
```

```
// metadata type name is X`1
.class public … X`1< … T> … { … }
```

```
generic<typename T, typename U>
public ref class X {
public:
   ref class Y { … };
   generic<typename A>
   ref class Z { … };
};
```

```
// metadata type name is X`2
.class public … X`2< … T, … U> … {

  // metadata type name is Y
  .class … nested public Y<( … T, … U> … { … }

  // metadata type name is Z`1
  .class … nested public Z`1<( …  T, … U, …  A> … { … }
}
```

end example]

Annex A. Grammar

A.1 Keywords

typedef-name:
    identifier

namespace-name:
    original-namespace-name
    namespace-alias

original-namespace-name:
    identifier

namespace-alias:
    identifier

class-name:
    identifier
    template-id

enum-name:
    identifier

template-name:
    identifier

property-or-event-name:
    identifier
    `default`

A.2 Lexical conventions

hex-quad:
    hexadecimal-digit *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit*

universal-character-name:
    `\u` *hex-quad*
    `\U` *hex-quad* *hex-quad*

preprocessing-token:
    header-name
    identifier
    pp-number
    character-literal
    string-literal
    preprocessing-op-or-punc
    each non-white-space character that cannot be one of the above

token
    identifier
    keyword
    literal
    operator
    punctuator

header-name:
<h-char-sequence>
"q-char-sequence"

h-char-sequence:
h-char
h-char-sequence h-char

h-char:
any member of the source character set except new-line and >

q-char-sequence
q-char
q-char-sequence q-char

q-char:
any member of the source character set except new-line and "

pp-number:
digit
`.` *digit*
pp-number digit
pp-number nondigit
pp-number `e` *sign*
pp-number `E` *sign*
pp-number `.`

identifier:
nondigit
identifier nondigit
identifier digit

nondigit: one of
universal-character-name

```
_   a   b   c   d   e   f   g   h   i   j   k   l   m
    n   o   p   q   r   s   t   u   v   w   x   y   z
    A   B   C   D   E   F   G   H   I   J   K   L   M
    N   O   P   Q   R   S   T   U   V   W   X   Y   Z
```

digit: one of

```
0   1   2   3   4   5   6   7   8   9
```

preprocessing-op-or-punc: one of

```
{       }       [       ]       #       ##      (       )
<:      :>      <%      %>      %:      %:%:    ;       :       ...
new     delete  ?       ::      .       .*
+       -       *       /       %       ^       &       |       ~
!       =       <       >       +=      -=      *=      /=      %=
^=      &=      |=      <<      >>      >>=     <<=     ==      !=
<=      >=      &&      ||      ++      --      ,       ->*     ->
and     and_eq  bitand  bitor   compl   not     not_eq
or      or_eq   xor     xor_eq
```

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal
null-literal

integer-literal:
decimal-literal *integer-suffix*$_{opt}$
octal-literal *integer-suffix*$_{opt}$
hexadecimal-literal *integer-suffix*$_{opt}$

decimal-literal:
nonzero-digit
decimal-literal *digit*

octal-literal:
`0`
octal-literal *octal-digit*

hexadecimal-literal:
`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-literal *hexadecimal-digit*

nonzero-digit: one of
`1` `2` `3` `4` `5` `6` `7` `8` `9`

octal-digit: one of
`0` `1` `2` `3` `4` `5` `6` `7`

hexadecimal-digit: one of
`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`
`a` `b` `c` `d` `e` `f`
`A` `B` `C` `D` `E` `F`

integer-suffix:
unsigned-suffix *long-suffix*$_{opt}$
unsigned-suffix *long-long-suffix*$_{opt}$
long-suffix *unsigned-suffix*$_{opt}$
long-long-suffix *unsigned-suffix*$_{opt}$

unsigned-suffix: one of
`u` `U`

long-suffix: one of
`l` `L`

long-long suffix: one of
`ll` `LL`

character-literal:
`'`*c-char-sequence*`'`
`L'`*c-char-sequence*`'`

c-char-sequence:
c-char
c-char-sequence *c-char*

c-char:
any member of the source character set except the single-quote `'`, backslash \, or new-line
character
escape-sequence
universal-character-name

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of
`\'` `\"` `\?` `\\`
`\a` `\b` `\f` `\n` `\r` `\t` `\v`

octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit* *octal-digit*
`\` *octal-digit* *octal-digit* *octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence *hexadecimal-digit*

floating-literal:
fractional-constant *exponent-part*$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence *exponent-part* *floating-suffix*$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ `.` *digit-sequence*
digit-sequence `.`

exponent-part:
`e` *sign*$_{opt}$ *digit-sequence*
`E` *sign*$_{opt}$ *digit-sequence*

sign: one of
`+` `-`

digit-sequence:
digit
digit-sequence *digit*

floating-suffix: one of
`f` `l` `F` `L`

string-literal:
"*s-char-sequence*$_{opt}$"
`L`"*s-char-sequence*$_{opt}$"

s-char-sequence:
s-char
s-char-sequence *s-char*

s-char:
any member of the source character set except the double-quote ", backslash \, or new-line
character
escape-sequence
universal-character-name

boolean-literal:
`false`
`true`

null-literal:
`nullptr`

A.3 Basic concepts

translation-unit:
declaration-seq$_{opt}$

A.4 Expressions

primary-expression:
literal
`this`
`(` *expression* `)`
id-expression

id-expression:
unqualified-id
qualified-id

unqualified-id:
identifier
operator-function-id
conversion-function-id
`~` *class-name*
`!` *class-name*
template-id
generic-id
`default`

qualified-id:
`::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id*
`::` *identifier*
`::` *operator-function-id*
`::` *template-id*

nested-name-specifier:
class-or-namespace-name `::` *nested-name-specifier*$_{opt}$
class-or-namespace-name `::` *template* *nested-name-specifier*

class-or-namespace-name:
class-name
namespace-name
property-or-event-name

postfix-expression:
primary-expression
postfix-expression `[` *expression-list* `]`
postfix-expression `(` *expression-list*$_{opt}$ `)`
simple-type-specifier `(` *expression-list*$_{opt}$ `)`
`typename` `::`$_{opt}$ *nested-name-specifier* *identifier* `(` *expression-list*$_{opt}$ `)`
`typename` `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id* `(` *expression-list*$_{opt}$ `)`
postfix-expression `.` *template*$_{opt}$ *id-expression*
postfix-expression `->` *template*$_{opt}$ *id-expression*
postfix-expression `.` *pseudo-destructor-name*
postfix-expression `->` *pseudo-destructor-name*
postfix-expression `++`
postfix-expression `--`
dynamic_cast `<` *type-id* `>` `(` *expression* `)`
`static_cast` `<` *type-id* `>` `(` *expression* `)`
`reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
`const_cast` `<` *type-id* `>` `(` *expression* `)`
`typeid` `(` *expression* `)`
`typeid` `(` *type-id* `)`
`typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier* *identifier* `::` `typeid`
`typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id* `::` `typeid`

expression-list:
assignment-expression
expression-list , *assignment-expression*

pseudo-destructor-name:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* :: ~ *type-name*
::$_{opt}$ *nested-name-specifier* template *template-id* :: ~ *type-name*
::$_{opt}$ *nested-name-specifier*$_{opt}$ ~ *type-name*

unary-expression:
postfix-expression
++ *cast-expression*
-- *cast-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-id*)
new-expression
delete-expression

unary-operator: one of
* & + - ! ~

new-expression:
::$_{opt}$ new *new-placement*$_{opt}$ *new-type-id new-initializer*$_{opt}$
::$_{opt}$ new *new-placement*$_{opt}$ (*type-id*) *new-initializer*$_{opt}$
gcnew *type-specifier-seq new-initializer*$_{opt}$ *array-init*$_{opt}$

new-placement:
(*expression-list*)

new-type-id:
type-specifier-seq new-declarator$_{opt}$

new-declarator:
ptr-operator new-declarator$_{opt}$
direct-new-declarator

direct-new-declarator:
[*expression*]
direct-new-declarator [*constant-expression*]

new-initializer:
(*expression-list*$_{opt}$)

array-init:
{ *initializer-list* ,$_{opt}$ }
{ }

delete-expression:
::$_{opt}$ delete *cast-expression*
::$_{opt}$ delete [ ] *cast-expression*

cast-expression:
unary-expression
(*type-id*) *cast-expression*

pm-expression:
cast-expression
pm-expression .* *cast-expression*
pm-expression ->* *cast-expression*

multiplicative-expression:
pm-expression
multiplicative-expression `*` *pm-expression*
multiplicative-expression `/` *pm-expression*
multiplicative-expression `%` *pm-expression*

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*

relational-expression:
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

equality-expression:
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

and-expression:
equality-expression
and-expression `&` *equality-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression `^` *and-expression*

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression `|` *exclusive-or-expression*

logical-and-expression:
inclusive-or-expression
logical-and-expression `&&` *inclusive-or-expression*

logical-or-expression:
logical-and-expression
logical-or-expression `||` *logical-and-expression*

conditional-expression:
logical-or-expression
logical-or-expression `?` *expression* `:` *assignment-expression*

assignment-expression:
conditional-expression
logical-or-expression *assignment-operator* *assignment-expression*
throw-expression

assignment-operator: one of
`=` `*=` `/=` `%=` `+=` `-=` `>>=` `<<=` `&=` `^=` `|=`

expression:
    assignment-expression
    expression `,` *assignment-expression*

constant-expression:
    conditional-expression

A.5 Statements

statement:
    labeled-statement
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    jump-statement
    declaration-statement
    try-block

labeled-statement:
    identifier `:` *statement*
    `case` *constant-expression* `:` *statement*
    `default :` *statement*

expression-statement:
    expression$_{opt}$ `;`

compound-statement:
    `{` *statement-seq*$_{opt}$ `}`

statement-seq:
    statement
    statement-seq *statement*

selection-statement:
    `if (` *condition* `)` *statement*
    `if (` *condition* `)` *statement* `else` *statement*
    `switch (` *condition* `)` *statement*

condition:
    expression
    type-specifier-seq *declarator* `=` *assignment-expression*

iteration-statement:
    `while (` *condition* `)` *statement*
    `do` *statement* *while* `(` *expression* `) ;`
    `for (` *for-init-statement* *condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
    `foreach (` *type-specifier-seq* *declarator* `in` *assignment-expression* `)` *statement*

for-init-statement:
    expression-statement
    simple-declaration

jump-statement:
    `break ;`
    `continue ;`
    `return` *expression*$_{opt}$ `;`
    `goto` *identifier* `;`

declaration-statement:
    block-declaration

A.6 Declarations

declaration-seq:
    declaration
    declaration-seq declaration

declaration:
    block-declaration
    function-definition
    template-declaration
    generic-declaration
    explicit-instantiation
    explicit-specialization
    linkage-specification
    namespace-definition

block-declaration:
    simple-declaration
    asm-definition
    namespace-alias-definition
    using-declaration
    using-directive

simple-declaration:
    attributes$_{opt}$ *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$;

decl-specifier:
    storage-class-specifier
    type-specifier
    function-specifier
    `friend`
    `typedef`

decl-specifier-seq:
    decl-specifier-seq$_{opt}$ *decl-specifier*

storage-class-specifier:
    `auto`
    `register`
    `static`
    `extern`
    `mutable`

function-specifier:
    `inline`
    `virtual`
    `explicit`

typedef-name:
    identifier

type-specifier:
    simple-type-specifier
    class-specifier
    enum-specifier
    elaborated-type-specifier
    cv-qualifier
    delegate-specifier

simple-type-specifier:
    ::$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*
    ::$_{opt}$ *nested-name-specifier* `template` *template-id*
    `char`
    `wchar_t`
    `bool`
    `short`
    `int`
    `long`
    `signed`
    `unsigned`
    `float`
    `double`
    `void`

type-name:
    class-name
    enum-name
    typedef-name

elaborated-type-specifier:
    attributes$_{opt}$ *class-key* ::$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
    attributes$_{opt}$ *class-key* ::$_{opt}$ *nested-name-specifier*$_{opt}$ *template*$_{opt}$ *template-id*
    attributes$_{opt}$ *enum-key* ::$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
    attributes$_{opt}$ `typename` ::$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
    attributes$_{opt}$ `typename` ::$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id*

enum-name:
    identifier

enum-specifier:
    attributes$_{opt}$ *top-level-visibility*$_{opt}$ *enum-key* *identifier*$_{opt}$ *enum-base*$_{opt}$
        `{` *enumerator-list*$_{opt}$ `}`

enum-key:
    `enum`
    `enum class`
    `enum struct`

enum-base:
    `:` *type-specifier-seq*

enumerator-list:
    enumerator-definition
    enumerator-list `,` *enumerator-definition*

enumerator-definition:
    enumerator
    enumerator `=` *constant-expression*

enumerator:
    attributes$_{opt}$ *identifier*

namespace-name:
    original-namespace-name
    namespace-alias

original-namespace-name:
    identifier

namespace-definition:
named-namespace-definition
unnamed-namespace-definition

named-namespace-definition:
original-namespace-definition
extension-namespace-definition

original-namespace-definition:
`namespace` *identifier* `{` *namespace-body* `}`

extension-namespace-definition:
`namespace` *original-namespace-name* `{` *namespace-body* `}`

unnamed-namespace-definition:
`namespace` `{` *namespace-body* `}`

namespace-body:
declaration-seq$_{opt}$

namespace-alias:
identifier

namespace-alias-definition:
namespace identifier `=` *qualified-namespace-specifier* `;`

qualified-namespace-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name*

using-declaration:
`using` `typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
`using` `::` *unqualified-id* `;`

using-directive:
`using` *namespace* `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name* `;`

asm-definition:
`asm` `(` *string-literal* `)` `;`

linkage-specification:
`extern` *string-literal* `{` *declaration-seq*$_{opt}$ `}`
`extern` *string-literal declaration*

A.7 Declarators

init-declarator-list:
init-declarator
init-declarator-list `,` *init-declarator*

init-declarator:
declarator initializer$_{opt}$

declarator:
direct-declarator
ptr-operator declarator

direct-declarator:
declarator-id
direct-declarator `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
exception-specification$_{opt}$
direct-declarator `[` *constant-expression*$_{opt}$ `]`
`(` *declarator* `)`

ptr-operator:
    `*` *cv-qualifier-seq*$_{opt}$
    `^` *cv-qualifier-seq*$_{opt}$
    `&`
    `%`
    `::`$_{opt}$ *nested-name-specifier* `*` *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
    cv-qualifier *cv-qualifier-seq*$_{opt}$

cv-qualifier:
    `const`
    `volatile`

declarator-id:
    id-expression
    `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*

type-id:
    type-specifier-seq *abstract-declarator*$_{opt}$

type-specifier-seq:
    type-specifier *type-specifier-seq*$_{opt}$

abstract-declarator:
    ptr-operator *abstract-declarator*$_{opt}$
    direct-abstract-declarator

direct-abstract-declarator:
    direct-abstract-declarator$_{opt}$
    `(` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$
    direct-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]`
    `(` *abstract-declarator* `)`

parameter-declaration-clause:
    parameter-declaration-list$_{opt}$ `...`$_{opt}$
    parameter-declaration-list `,` `...`
    parameter-array
    parameter-declaration-list `,` *parameter-array*

parameter-declaration-list:
    parameter-declaration
    parameter-declaration-list `,` *parameter-declaration*

parameter-declaration:
    attributes$_{opt}$ *decl-specifier-seq* *declarator*
    attributes$_{opt}$ *decl-specifier-seq* *declarator* `=` *assignment-expression*
    attributes$_{opt}$ *decl-specifier-seq* *abstract-declarator*$_{opt}$
    attributes$_{opt}$ *decl-specifier-seq* *abstract-declarator*$_{opt}$ `=` *assignment-expression*

parameter-array:
    attributes$_{opt}$ `...` *parameter-declaration*

function-definition:
    attributes$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* *function-modifiers*$_{opt}$ *override-specifier*$_{op}$
        ctor-initializer$_{opt}$ *function-body*
    attributes$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* *function-modifiers*$_{opt}$ *override-specifier*$_{opt}$
        function-try-block

function-body:
    compound-statement

initializer:
= *initializer-clause*
(*expression-list*)

initializer-clause:
assignment-expression
{ *initializer-list* ,$_{opt}$ }
{ }

initializer-list:
initializer-clause
initializer-list , *initializer-clause*

A.8 Classes

class-name:
identifier
template-id

class-specifier:
attributes$_{opt}$ *top-level-visibility*$_{opt}$ *class-head* { *member-specification*$_{opt}$ }

top-level-visibility:
public
private

class-head:
class-key *identifier*$_{opt}$ *class-modifiers*$_{opt}$ *base-clause*$_{opt}$
class-key *nested-name-specifier* *identifier* *class-modifiers*$_{opt}$ *base-clause*$_{opt}$
class-key *nested-name-specifier*$_{opt}$ *template-id* *class-modifiers*$_{opt}$ *base-clause*$_{opt}$

class-key:
class
struct
union
ref class
ref struct
value class
value struct
interface class
interface struct

class-modifiers:
class-modifiers$_{opt}$ *class-modifier*

class-modifier:
abstract
sealed

member-specification:
member-declaration *member-specification*$_{opt}$
access-specifier : *member-specification*$_{opt}$

member-declaration:
attributes$_{opt}$ *initonly-or-literal*$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$ `;`
function-definition `;`$_{opt}$
`::`$_{opt}$ *nested-name-specifier* *template*$_{opt}$ *unqualified-id* `;`
using-declaration
template-declaration
generic-declaration
delegate-specifier
event-definition
property-definition

initonly-or-literal:
`initonly`
`literal`

member-declarator-list:
member-declarator
member-declarator-list `,` *member-declarator*

member-declarator:
declarator *function-modifiers*$_{opt}$ *override-specifier*$_{opt}$
declarator *constant-initializer*$_{opt}$
identifier$_{opt}$ `:` *constant-expression*

function-modifiers:
function-modifiers$_{opt}$ *function-modifier*

function-modifier:
`abstract`
`new`
`override`
`sealed`

override-specifier:
`=` *overridden-name-list*
pure-specifier

overridden-name-list:
id-expression
overridden-name-list `,` *id-expression*

pure-specifier:
`=` `0`

constant-initializer:
`=` *constant-expression*

A.9 Properties and events

property-definition:
attributes$_{opt}$ *property-modifiers*$_{opt}$ `property` *type-specifier-seq* *declarator* *property-indexes*$_{opt}$
`{` *accessor-specification* `}`
attributes$_{opt}$ *property-modifiers*$_{opt}$ `property` *type-specifier-seq* *declarator* `;`

property-modifiers:
property-modifiers$_{opt}$ *property-modifier*

property-modifier:
`static`
`virtual`

property-indexes:
`[` *property-index-parameter-list* `]`

property-index-parameter-list:
type-id
property-index-parameter-list `,` *type-id*

accessor-specification:
accessor-declaration *accessor-specification*$_{opt}$
access-specifier `:` *accessor-specification*$_{opt}$

accessor-declaration:
attributes$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$ `;`
function-definition

event-definition:
attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *event-type* *identifier*
`{` *accessor-specification* `}`
attributes$_{opt}$ *event-modifiers*$_{opt}$ `event` *event-type* *identifier* `;`

event-modifiers:
event-modifiers$_{opt}$ *event-modifier*

event-modifier:
`static`
`virtual`

event-type:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* `^`$_{opt}$
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ `template` *template-id* `^`

A.10 Derived classes

base-clause:
`:` *base-specifier-list*

base-specifier-list:
base-specifier
base-specifier-list `,` *base-specifier*

base-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
`virtual` *access-specifier*$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
access-specifier `virtual`$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

access-specifier:
`private`
`protected`
`public`
`internal`
`protected public`
`public protected`
`private protected`
`protected private`

A.11 Special member functions

conversion-function-id:
`operator` *conversion-type-id*

conversion-type-id:
type-specifier-seq *conversion-declarator*$_{opt}$

conversion-declarator:
ptr-operator *conversion-declarator*$_{opt}$

ctor-initializer:
`:` *mem-initializer-list*

mem-initializer-list:
mem-initializer
mem-initializer `,` *mem-initializer-list*

mem-initializer:
mem-initializer-id `(` *expression-list*$_{opt}$ `)`

mem-initializer-id:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
identifier

A.12 Overloading

operator-function-id:
`operator` *operator*
`operator` *operator* `<` *template-argument-list*$_{opt}$ `>`

operator: one of

```
new    delete     new[]      delete[]
+      -     *      /      %      ^      &      |      ~
!      =     <      >      +=     -=     *=     /=     %=
^=     &=    |=     <<     >>     >>=    <<=    ==     !=
<=     >=    &&     ||     ++     --     ,      ->*    ->
()     []
```

A.13 Delegates

delegate-specifier:
attributes$_{opt}$ *top-level-visibility*$_{opt}$ `delegate` *type-specifier-seq* *declarator* `;`

A.14 Templates

template-declaration:
`export`$_{opt}$ `template` `<` *template-parameter-list* `>` *declaration*

template-parameter-list:
template-parameter
template-parameter-list `,` *template-parameter*

template-parameter:
type-parameter
parameter-declaration

type-parameter:
`class` *identifier*$_{opt}$
`class` *identifier*$_{opt}$ `=` *type-id*
`typename` *identifier*$_{opt}$
`typename` *identifier*$_{opt}$ `=` *type-id*
`template` `<` *template-parameter-list* `>` `class` *identifier*$_{opt}$
`template` `<` *template-parameter-list* `>` `class` *identifier*$_{opt}$ `=` *id-expression*

template-id:
template-name `<` *template-argument-list*$_{opt}$ `>`

template-name:
identifier

template-argument-list:
template-argument
template-argument-list , *template-argument*

template-argument:
assignment-expression
type-id
id-expression

explicit-instantiation:
`template` *declaration*

explicit-specialization:
`template < >` *declaration*

A.15 Generics

generic-declaration:
`generic <` *generic-parameter-list* `>` *constraint-clause-list*$_{opt}$ *declaration*

generic-parameter-list:
generic-parameter
generic-parameter-list , *generic-parameter*

generic-parameter:
attributes$_{opt}$ `class` *identifier*
attributes$_{opt}$ `typename` *identifier*

generic-id:
generic-name `<` *generic-argument-list* `>`

generic-name:
identifier
operator-function-id

generic-argument-list:
generic-argument
generic-argument-list , *generic-argument*

generic-argument:
type-id

constraint-clause-list:
constraint-clause-list$_{opt}$ *constraint-clause*

constraint-clause:
`where` *identifier* : *constraint-item-list*

constaint-item-list:
constraint-item
constraint-item-list , *constraint-item*

constraint-item:
type-id
`ref class`
`ref struct`
`value class`
`value struct`
`gcnew ( )`

A.16 Exception handling

try-block:
    `try` *compound-statement* *handler-seq*
    `try` *compound-statement* *finally-clause*
    `try` *compound-statement* *handler-seq* *finally-clause*

function-try-block:
    `try` *ctor-initializer*$_{opt}$ *function-body* *handler-seq*
    `try` *ctor-initializer*$_{opt}$ *function-body* *finally-clause*
    `try` *ctor-initializer*$_{opt}$ *function-body* *handler-seq* *finally-clause*

handler-seq:
    handler *handler-seq*$_{opt}$

handler:
    catch `(` *exception-declaration* `)` *compound-statement*

exception-declaration:
    type-specifier-seq *declarator*
    type-specifier-seq *abstract-declarator*
    type-specifier-seq
    `...`

finally-clause:
    `finally` *compound-statement*

throw-expression:
    `throw` *assignment-expression*$_{opt}$

exception-specification:
    `throw` `(` *type-id-list*$_{opt}$ `)`

type-id-list:
    type-id
    type-id-list `,` *type-id*

A.17 Attributes

attributes:
    attribute-sections

attribute-sections:
    attribute-sections$_{opt}$ *attribute-section*

attribute-section:
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`

attribute-target-specifier:
    attribute-target `:`

attribute-target:
    `assembly`
    `class`
    `constructor`
    `delegate`
    `enum`
    `event`
    `field`
    `interface`
    `method`
    `parameter`
    `property`
    `returnvalue`
    `struct`

attribute-list:
    attribute
    attribute-list `,` *attribute*

attribute:
    attribute-name *attribute-arguments*$_{opt}$

attribute-name:
    type-name

attribute-arguments:
    `(` *positional-argument-list*$_{opt}$ `)`
    `(` *positional-argument-list* `,` *named-argument-list* `)`
    `(` *named-argument-list* `)`

positional-argument-list:
    positional-argument
    positional-argument-list `,` *positional-argument*

positional-argument:
    attribute-argument-expression

named-argument-list:
    named-argument
    named-argument-list `,` *named-argument*

named-argument:
    identifier `=` *attribute-argument-expression*

attribute-argument-expression:
    assignment-expression

A.18 Preprocessing directives

preprocessing-file:
    group$_{opt}$

group:
    group-part
    group *group-part*

group-part:
    pp-tokens$_{opt}$ *new-line*
    if-section
    control-line

if-section:
    if-group *elif-groups*$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
    `#` `if` *constant-expression* *new-line* *group*$_{opt}$
    `#` `ifdef` *identifier* *new-line* *group*$_{opt}$
    `#` `ifndef` *identifier* *new-line* *group*$_{opt}$

elif-groups:
    elif-group
    elif-groups *elif-group*

elif-group:
    `#` `elif` *constant-expression* *new-line* *group*$_{opt}$

else-group:
    `#` `else` *new-line* *group*$_{opt}$

endif-line:
    `#` `endif` *new-line*

control-line:
    `#` `include` *pp-tokens* *new-line*
    `#` `using` *pp-tokens* *new-line*
    `#` `define` *identifier* *replacement-list* *new-line*
    `#` `define` *identifier* *lparen* *identifier-list*$_{opt}$ `)` *replacement-list* *new-line*
    `#` `undef` *identifier* *new-line*
    `#` `line` *pp-tokens* *new-line*
    `#` `error` *pp-tokens*$_{opt}$ *new-line*
    `#` `pragma` *pp-tokens*$_{opt}$ *new-line*
    `#` *new-line*

lparen:
    the left-parenthesis character without preceding white-space

replacement-list:
    pp-tokens$_{opt}$

pp-tokens:
    preprocessing-token
    pp-tokens *preprocessing-token*

new-line:
    the new-line character

Annex B. Verifiable code

[*Note*: Reserved for future use. *end note*]

Annex C. Documentation comments

This annex is informative.

C.1 Introduction

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements they precede. Such comments are single-line comments that start with exactly three slashes (`///`). They shall immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or function) that they annotate. Attribute sections are considered part of declarations, so documentation comments shall precede attributes applied to a type or member.

Alternatively, comments (possibly multi-line) that start with a slash and exactly two asterisks may also contain XML document comments.

These comments may only be applied to CLI class types and members within those types. While processing such comments, if they are applied to unsupported types, the compiler shall issue a warning.

Documentation comments in a header are processed only if that header were included using the "…" form of `#include`.

Syntax:

single-line-doc-comment:
`///` *intput-characters*$_{opt}$

delimited-doc-comment:
`/**` *delimited-comment-characters*$_{opt}$ `*/`

In a *single-line-doc-comment*, if there is a *white-space* character following the `///` characters on each of the *single-line-doc-comment*s adjacent to the current *single-line-doc-comment*, then that one *white-space* character is not included in the XML output.

In a *delimited-doc-comment*, if the first non-*white-space* character on the second line is an *asterisk* and the same pattern of optional *white-space* characters and an *asterisk* character is repeated at the beginning of each of the lines within the *delimited-doc-comment*, then the characters of the repeated pattern are not included in the XML output. The pattern can include *white-space* character after, as well as before, the *asterisk* character.

Example:

```
/**
<remarks>
Class <c>Point</c> models a point in a two-dimensional plane.
</remarks>
*/
public ref class Point {
public:
   /// <remarks>Method <c>Draw</c> renders the point.</remarks>
   void Draw() { /*...*/ }
};
```

The text within documentation comments shall be well-formed according to the rules of XML (http://www.w3.org/TR/REC-xml). If the XML is ill-formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in §C.2. Some of the recommended tags have special meanings:

- The <param> tag is used to describe parameters. If such a tag is used, the documentation generator shall verify that the specified parameter exists and that all parameters are described in documentation comments. If such verification fails, the documentation generator issues a warning.
- The cref attribute can be attached to any tag to provide a reference to a code element. The documentation generator shall verify that this code element exists. If the verification fails, the documentation generator issues a warning. When looking for a name described in a cref attribute, the documentation generator shall respect namespace visibility according to using statements appearing within the source code.
- The <summary> tag is intended to be used by a documentation viewer to display additional information about a type or member.

Note carefully that the documentation file does not provide full information about the type and members (for example, it does not contain any type information). To get such information about a type or member, the documentation file shall be used in conjunction with reflection on the actual type or member.

C.2 Recommended tags

The documentation generator shall accept and process any tag that is valid according to the rules of XML. The following tags provide commonly used functionality in user documentation. (Of course, other tags are possible.)

Tag	Section	Purpose
<c>	§C.2.1	Set text in a code-like font
<code>	§C.2.2	Set one or more lines of source code or program output
<example>	§C.2.3	Indicate an example
<exception>	§C.2.4	Identifies the exceptions a function can throw
<list>	§C.2.5	Create a list or table
<para>	§C.2.6	Permit structure to be added to text
<param>	§C.2.7	Describe a parameter for a function or constructor
<paramref>	§C.2.8	Identify that a word is a parameter name
<permission>	§C.2.9	Document the security accessibility of a member
<remarks>	§C.2.10	Describe a type
<returns>	§C.2.11	Describe the return value of a function
<see>	§C.2.12	Specify a link
<seealso>	§C.2.13	Generate a See Also entry
<summary>	§C.2.14	Describe a member of a type
<typeparam>	§C.2.15	Describe a generic type parameter
<typeparamref>	§C.2.16	Identify that a word is a type parameter name
<value>	§C.2.17	Describe a property

C.2.1 <c>

This tag provides a mechanism to indicate that a fragment of text within a description should be set in a special font such as that used for a block of code. For lines of actual code, use <code> (§C.2.2).

Syntax:

```
<c>text to be set like code</c>
```

Example:

```
/// <remarks>
/// Class <c>Point</c> models a point in a two-dimensional plane.
/// </remarks>
ref class Point
{
   // ...
};
```

C.2.2 <code>

This tag is used to set one or more lines of source code or program output in some special font. For small code fragments in narrative, use <c> (§C.2.1).

Syntax:

```
<code>source code or program output</code>
```

Example:

```
/// <summary>
///   Changes the Point's location by the given x- and y-offsets.
/// <example>
///   The following code:
///   <code>
///     Point p(3,5);
///     p.Translate(-1,3);
///   </code>
///   results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
void Translate(int xord, int yord) {
   X += xord;
   Y += yord;
}
```

C.2.3 <example>

This tag allows example code within a comment, to specify how a function or other library member may be used. Ordinarily, this would also involve use of the tag <code> (§C.2.2) as well.

Syntax:

```
<example>description</example>
```

Example:

See <code> (§C.2.2) for an example.

C.2.4 <exception>

This tag provides a way to document the exceptions a function can throw.

Syntax:

```
<exception cref="member">description</exception>
```

where

```
cref="member"
```

The name of a member. The documentation generator checks that the given member exists and translates *member* to the canonical element name in the documentation file.

```
description
```

A description of the circumstances in which the exception is thrown.

Example:

```
public ref class DataBaseOperations
{
   /// <exception cref="MasterFileFormatCorruptException">…</exception>
   /// <exception cref="MasterFileLockedOpenException">…</exception>
   static void ReadRecord(int flag) {
      if (flag == 1)
         throw new MasterFileFormatCorruptException();
      else if (flag == 2)
         throw new MasterFileLockedOpenException();
      // …
   }
};
```

C.2.5 <list>

This tag is used to create a list or table of items. It may contain a `<listheader>` block to define the heading row of either a table or definition list. (When defining a table, only an entry for `term` in the heading need be supplied.)

Each item in the list is specified with an `<item>` block. When creating a definition list, both `term` and `description` shall be specified. However, for a table, bulleted list, or numbered list, only `description` need be specified.

Syntax:

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>description</description>
   </listheader>
   <item>
      <term>term</term>
      <description>description</description>
   </item>
   …
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

where

`term`

The term to define, whose definition is in `description`.

`description`

Either an item in a bullet or numbered list, or the definition of a `term`.

Example:

```
public ref class MyClass {
public:
   /// <remarks>
   ///   Here is an example of a bulleted list:
   ///   <list type="bullet">
   ///     <item>
   ///       <description>First item.</description>
   ///     </item>
   ///     <item>
   ///       <description>Second item.</description>
   ///     </item>
   ///   </list>
   /// </remarks>
   static void F() {
      // ...
   }
};
```

C.2.6 <para>

This tag is for use inside other tags, such as <remarks> (§C.2.10) or <returns> (§C.2.11), and permits structure to be added to text.

Syntax:

```
<para>content</para>
```

where

```
content
```

The text of the paragraph.

Example:

```
/// <summary>
///   <para>
///     This is the entry point of the Point class testing program.
///   </para>
///   <para>
///     This program tests each function and operator, and is intended
///     to be run after any non-trivial maintenance has been performed
///     on the Point class.
///   </para>
/// </summary>
int main() {
   // ...
}
```

C.2.7 <param>

This tag is used to describe a parameter for a function, constructor, or indexer.

Syntax:

```
<param name="name">description</param>
```

where

```
name
```

The name of the parameter.

```
description
```

A description of the parameter.

Example:

```
/// <summary>
///   This function changes the point's location to the given
coordinates.
/// </summary>
/// <param name="xord"><c>xord</c> is the new x-coordinate.</param>
/// <param name="yord"><c>yord</c> is the new y-coordinate.</param>
void Move(int xord, int yord) {
   X = xord;
   Y = yord;
}
```

C.2.8 <paramref>

This tag is used to indicate that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<paramref name="name"/>
```

where

`name`

The name of the parameter.

Example:

```
/// <summary>
///   This constructor initializes the new Point to
///   (<paramref name="xord"/>,<paramref name="yord"/>).
/// </summary>
/// <param name="xord">
///   <c>xord</c> is the new Point's x-coordinate.
/// </param>
/// <param name="yord">
///   <c>yord</c> is the new Point's y-coordinate.
/// </param>
Point(int xord, int yord) {
   X = xord;
   Y = yord;
}
```

C.2.9 <permission>

This tag allows the security accessibility of a member to be documented.

Syntax:

```
<permission cref="member">description</permission>
```

where

`cref="member"`

The name of a member. The documentation generator checks that the given code element exists and translates *member* to the canonical element name in the documentation file.

`description`

A description of the access to the member.

Example:

```
/// <permission cref="System::Security::PermissionSet">
///   Everyone can access this function.
/// </permission>
static void Test() {
   // ...
}
```

C.2.10 <remarks>

This tag is used to specify overview information about a type. (Use `<summary>` (§C.2.14) to describe the members of a type.)

Syntax:

```
<remarks>description</remarks>
```

where

`description`

The text of the remarks.

Example:

```
/// <remarks>
///   Class <c>Point</c> models a point in a two-dimensional plane.
/// </remarks>
public ref class Point
{
   // ...
};
```

C.2.11 <returns>

This tag is used to describe the return value of a function.

Syntax:

```
<returns>description</returns>
```

where

```
description
```

A description of the return value.

Example:

```
/// <summary>
///   Report a point's location as a string.
/// </summary>
/// <returns>
///   A string representing a point's location, in the form (x,y),
///   without any leading, trailing, or embedded whitespace.
/// </returns>
String^ ToString() override {
   return String::Format("({0},{1})", X, Y);
}
```

C.2.12 <see>

This tag allows a link to be specified within text. Use <seealso> (§C.2.13) to indicate text that is to appear in a See Also subclause.

Syntax:

```
<see cref="member"/>
```

where

```
cref="member"
```

The name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>
///   This function changes the point's location to the given
coordinates.
///   Use the <see cref="Translate"/> function to apply a relative
change.
/// </summary>
void Move(int xord, int yord) {
   X = xord;
   Y = yord;
}

/// <summary>
///   This function changes the point's location by the given offsets.
///   Use the <see cref="Move"/> function to directly set the
coordinates.
/// </summary>
void Translate(int xord, int yord) {
   X += xord;
   Y += yord;
}
```

C.2.13 <seealso>

This tag allows an entry to be generated for the See Also section. Use <see> (§C.2.12) to specify a link from within text.

Syntax:

```
<seealso cref="member"/>
```

where

```
cref="member"
```

The name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>
///   This function determines whether two Points have the same location.
/// </summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
bool Equals(Object^ o) override {
   // ...
}
```

C.2.14 <summary>

This tag can be used to describe a member for a type. Use `<remarks>` (§C.2.10) to describe the type itself.

Syntax:

```
<summary>description</summary>
```

where

```
description
```

A summary of the member.

Example:

```
/// <summary>
///   This constructor initializes the new Point to (0,0).
/// </summary>
Point() {
   // …
}
```

C.2.15 <typeparam>

This tag is used to describe a type parameter for a generic type or function.

Syntax:

```
<typeparam name="name">description</typeparam>
```

where

```
name
```

The name of the type parameter.

```
description
```

A description of the type parameter.

Example:

```
/// <summary>
///   A single linked list that stores unique elements.
/// </summary>
/// <typeparam name="T">Each element of the list is a
<c>T</c>.</typeparam>
generic<typename T>
ref class List {
   /* ... */
};
```

C.2.16 <typeparamref>

This tag is used to indicate that a word is a type parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<typeparamref name="name"/>
```

where

```
name
```

The name of the parameter.

C.2.17 <value>

This tag allows a property to be described.

Syntax:

```
<value>property description</value>
```

where

```
property description
```

A description for the property.

Example:

```
/// <value>
///   The point's x-coordinate.
/// </value>
property int X {
   int get() { return x; }
   void set(int value) { x = value; }
}
```

C.3 Processing the documentation file

The following information is intended for C++/CLI implementations targeting the CLI.

The documentation generator generates an ID string for each element in the source code that is tagged with a documentation comment. This ID string uniquely identifies a source element. A documentation viewer can use an ID string to identify the corresponding metadata/reflection item to which the documentation applies.

The documentation file is not a hierarchical representation of the source code; rather, it is a flat list with a generated ID string for each element.

C.3.1 ID string format

The documentation generator observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the string identifies the kind of member being documented, via a single character followed by a colon. The following kinds of members are defined:

Character	Description
E	Event
F	Field
M	Method (including constructors, destructors, finalizers, functions, and operators)
N	Namespace
P	Property (including indexers)

D	Typedef
T	Type (such as class, delegate, enum, interface, and struct)
!	Error string; the rest of the string provides information about the error. For example, the documentation generator generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the element, starting at the root of the namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by NUMBER SIGN # (U+0023) characters. (It is assumed that no element has this character in its name.)
- For functions and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows: Arguments are represented by their fully qualified name. For example, `int` is `System.Int32`, and so on. Tracking reference arguments have an `@` following their type name. Arguments passed by value or via param arrays have no special notation. Arguments that are CLI arrays are represented as `[` *lowerbound* `:` *size* `,` … `,` *lowerbound* `:` *size* `]` where the number of commas is the rank less one, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the "`:`" is omitted as well. Jagged arrays are represented by one "`[]`" per level. Arguments that have pointer types other than void are represented using a `*` following the type name. A void pointer is represented using a type name of `System.Void`.

C.3.2 ID string examples

The following examples each show a fragment of C++ code, along with the ID string produced from each source element capable of having a documentation comment:

- Types are represented using their fully qualified name.

```
enum class Color { Red, Blue, Green };

namespace Acme {
      interface class IProcess { /*...*/ };
      value class ValueType { /*...*/ };
      ref class Widget : IProcess {
      public:
         ref class NestedClass { /*...*/ };
         interface class IMenuItem { /*...*/ };
         delegate void Del(int i);
         enum class Direction { North, South, East, West };
      };
}
```

```
"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
```

- Fields are represented by their fully qualified name.

```
namespace Acme {
   value class ValueType {
   private:
         int total;
   };
```

```
      ref class Widget: IProcess {
      public:
         ref class NestedClass {
         private:
            int value;
         };

       private:
         String^ message;
         static Color^ defaultColor;
         literal double PI = 3.14159;
         initonly double monthlyAverage;
         array<long>^ array1;
         array<Widget^,2>^ array2;
         int *pCount;
         float **ppValues;
      };
   }
```

```
"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- Constructors.

```
   namespace Acme {
      ref class Widget : IProcess {
         static Widget() { /*...*/ }
      public:
         Widget() { /*...*/ }
         Widget(String^ s) { /*...*/ }
      };
   }
```

```
"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- Finalizers.

```
   namespace Acme {
      ref class Widget : IProcess   {
   protected:
         !Widget() { /*...*/ }
      };
   }
```

```
"M:Acme.Widget.Finalize"
```

- Methods.

```
   namespace Acme {
      value class ValueType {
      public:
         void M(int i) { /*...*/ }
      };

      ref class Widget : IProcess {
      public:
       ref class NestedClass {
         public:
          void M(int i) { /*...*/ }
```

```
            };

            static void M0() { /*...*/ }
            void M1(wchar_t c, float% f, ValueType% v) { /*...*/ }
            void M2(array<short>^ x1, array<int,2>^ x2,
   array<array<int>^>^ x3)
               { /*...*/ }
            void M3(array<array<int>^> x3, array<array<Widget^,3>^>^ x4)
               { /*...*/ }
            void M4(wchar_t *pc, Color **pf) { /*...*/ }
            void M5(void *pv, array<array<double*,2>^ > pd) { /*...*/ }
            void M6(int i, ... array<Object^>^ args) { /*...*/ }
         };
      }
```

```
   "M:Acme.ValueType.M(System.Int32)"
   "M:Acme.Widget.NestedClass.M(System.Int32)"
   "M:Acme.Widget.M0"
   "M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
   "M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
   "M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
   "M:Acme.Widget.M4(System.Char*,Color**)"
   "M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
   "M:Acme.Widget.M6(System.Int32,System.Object[])"
```

- Properties and indexers.

```
      namespace Acme {
         ref class Widget : IProcess {
         public:
            property int Width {
               int get() { /*...*/ }
               void set(int value) { /*...*/ }
            }

            property int default[int] {
               int get(int i) { /*...*/ }
               void set(int i, int value) { /*...*/ }
            }

            property int default[String^, int] {
               int get(String^ s, int i) { /*...*/ }
               void set(String^ s, int i, int value) { /*...*/ }
            }
         };
      }
```

```
   "P:Acme.Widget.Width"
   "P:Acme.Widget.Item(System.Int32)"
   "P:Acme.Widget.Item(System.String,System.Int32)"
```

- Events.

```
      namespace Acme {
         ref class Widget : IProcess {
         public:
            event Del^ AnEvent;
         };
      }
```

```
"E:Acme.Widget.AnEvent"
```

- Unary operators. (The complete set of unary operator function names used is listed in Table 19-1: CLS-Compliant Unary Operators.)

```
      namespace Acme {
         ref class Widget : IProcess {
         public:
            static Widget^ operator+(Widget^ x) { /*...*/ }
```

```
      };
   }
```

```
"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

- Binary operators. (The complete set of binary operator function names used is listed in Table 19-2: CLS-Compliant Binary Operators.)

```
namespace Acme {
   ref class Widget : IProcess {
   public:
      static Widget^ operator+(Widget^ x1, Widget^ x2) { /*...*/ }
   };
}
```

```
"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

- Conversion operators have a trailing "~" followed by the return type.

```
namespace Acme {
   ref class Widget : IProcess {
   public:
      static explicit operator int(Widget^ x) { /*...*/ }
      static operator long long(Widget^ x) { /*...*/ }
   };
}
```

```
"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

C.4 An example

C.4.1 C++ source code

The following example shows the source code of a `Point` class:

```
namespace Graphics {
   /// <remarks>
   ///   Class <c>Point</c> models a point in a two-dimensional plane.
   /// </remarks>
   public ref class Point {
   public:
      /// <value>
      ///   The Point's x-coordinate.
      /// </value>
      property int X;

      /// <value>
      ///   The Points' y-coordinate.
      /// </value>
      property int Y;

      /// <summary>
      ///   This constructor initializes the new Point to (0,0).
      /// </summary>
      Point() {
         X = 0;
         Y = 0;
      }

      /// <summary>
      ///   This constructor initializes the new Point to
      ///   (<paramref name="xord"/>,<paramref name="yord"/>).
      /// </summary>
      /// <param name="xord">
      ///   <c>xord</c> is the new Point's x-coordinate.
      /// </param>
      /// <param name="yord">
      ///   <c>yord</c> is the new Point's y-coordinate.
      /// </param>
      Point(int xord, int yord) {
         X = xord;
         Y = yord;
      }

      /// <summary>
      ///   This function changes the point's location to the given
      ///   coordinates.
      /// </summary>
      /// <param name="xord">
      ///   <c>xord</c> is the new x-coordinate.
      /// </param>
      /// <param name="yord">
      ///   <c>yord</c> is the new y-coordinate.
      /// </param>
      /// <seealso cref="Translate"/>
      void Move(int xord, int yord) {
         X = xord;
         Y = yord;
      }

      /// <summary>
      ///   This function changes the point's location by the given
      ///   x- and y-offsets.
      /// </summary>
      /// <example>
      ///   The following code:
      ///   <code>
      ///     Point p(3,5);
      ///     p.Translate(-1,3);
      ///   </code>
      ///   results in <c>p</c>'s having the value (2,8).
      /// </example>
      /// <param name="xord">
      ///   <c>xord</c> is the relative x-offset.
```

```
/// </param>
/// <param name="yord">
///   <c>yord</c> is the relative y-offset.
/// </param>
/// <seealso cref="Move"/>
void Translate(int xord, int yord) {
   X += xord;
   Y += yord;
}

/// <summary>
///   This function determines whether two Points have the same
///   location.
/// </summary>
/// <param name="o">
///   <c>o</c> is the object to be compared to the current object.
/// </param>
/// <returns>
///   True if the Points have the same location; otherwise, false.
/// </returns>
/// <seealso cref="operator =="/>
/// <seealso cref="operator !="/>
bool Equals(Object^ o) override {
   Point^ p = dynamic_cast<Point^>(o);
   if (!p) return false;
   return (X == p->X) && (Y == p->Y);
}

/// <summary>
///   Computes the hash code for a Point.
/// </summary>
/// <returns>
///   A hash code computed from the x and y coordinates.
/// </returns>
int GetHashCode() override {
   return X ^ Y;
}

/// <summary>
///   Report a point's location as a string.
/// </summary>
/// <returns>
///   A string representing a point's location, in the form (x,y),
///   without any leading, training, or embedded whitespace.
/// </returns>
String^ ToString() override {
   return String::Format("({0},{1})", X, Y);
}

/// <summary>
///   This operator determines whether two Points have the same
///   location.
/// </summary>
/// <param name="p1">The first Point to be compared.</param>
/// <param name="p2">The second Point to be compared.</param>
/// <returns>
///   True if the Points have the same location; otherwise, false.
/// </returns>
/// <seealso cref="Equals"/>
/// <seealso cref="operator !="/>
static bool operator==(Point^ p1, Point^ p2) {
   if ((Object^)p1 == nullptr || (Object^)p2 == nullptr)
      return false;
   return (p1->X == p2->X) && (p1->Y == p2->Y);
}

/// <summary>
///   This operator determines whether two Points have the same
///   location.
```

```
            /// </summary>
            /// <param name="p1">The first Point to be compared.</param>
            /// <param name="p2">The second Point to be compared.</param>
            /// <returns>
            ///   True if the Points do not have the same location;
            ///   otherwise, false.
            /// </returns>
            /// <seealso cref="Equals"/>
            /// <seealso cref="operator =="/>
            static bool operator!=(Point^ p1, Point^ p2) {
               return !(p1 == p2);
            }
         };
      }
```

C.4.2 Resulting XML

Here is the output produced by one documentation generator when given the source code for class `Point`, shown above:

```
<?xml version="1.0"?>
<doc>
 <assembly>
    Point
 </assembly>
 <members>
  <member name="T:Graphics.Point">
   <remarks>
    Class <c>Point</c> models a point in a two-dimensional plane.
   </remarks>
  </member>

  <member name="M:Graphics.Point.get_X">
   <value>
   The Point's x-coordinate.
   </value>
  </member>

  <member name="M:Graphics.Point.get_Y">
   <value>
   The Points' y-coordinate.
   </value>
  </member>

  <member name="M:Graphics.Point.#ctor">
   <summary>
   This constructor initializes the new Point to (0,0).
   </summary>
 </member>

 <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
  <summary>
  This constructor initializes the new Point to
  (<paramref name="xord"/>,<paramref name="yord"/>).
  </summary>

  <param name="xord">
   <c>xord</c> is the new Point's x-coordinate.
  </param>

  <param name="yord">
   <c>yord</c> is the new Point's y-coordinate.
```

```
 </param>
</member>

<member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
 <summary>
 This function changes the point's location to the given coordinates.
 </summary>

 <param name="xord">
   <c>xord</c> is the new x-coordinate.
 </param>

 <param name="yord">
   <c>yord</c> is the new y-coordinate.
 </param>

 <seealso cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
</member>

<member name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
 <summary>
 This function changes the point's location by the given x- and y-offsets.
 </summary>

<example>
 The following code:
 <code>
    Point p(3,5);
    p.Translate(-1,3);
 </code>
 results in <c>p</c>'s having the value (2,8).
</example>

<param name="xord">
  <c>xord</c> is the relative x-offset.
</param>

<param name="yord">
   <c>yord</c> is the relative y-offset.
</param>
<seealso cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
</member>

<member name="M:Graphics.Point.Equals(System.Object)">
  <summary>
  This function determines whether two Points have the same location.
</summary>

<param name="o">
  <c>o</c> is the object to be compared to the current object.
</param>

<returns>
  True if the Points have the same location; otherwise, false.
</returns>

<seealso cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
<seealso cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
</member>
```

```
<member name="M:Graphics.Point.GetHashCode">
  <summary>
  Computes the hash code for a Point.
  </summary>

  <returns>
  A hash code computed from the x and y coordinates.
  </returns>
 </member>

 <member name="M:Graphics.Point.ToString">
  <summary>
  Report a point's location as a string.
  </summary>

  <returns>
  A string representing a point's location, in the form (x,y),
  without any leading, training, or embedded whitespace.
  </returns>
</member>

<member name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
  <summary>
  This operator determines whether two Points have the same location.
  </summary>

  <param name="p1">The first Point to be compared.</param>
  <param name="p2">The second Point to be compared.</param>

  <returns>
  True if the Points have the same location; otherwise, false.
  </returns>

  <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
  <seealso cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
</member>

<member name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
 <summary>
  This operator determines whether two Points have the same location.
 </summary>

 <param name="p1">The first Point to be compared.</param>
 <param name="p2">The second Point to be compared.</param>

 <returns>
  True if the Points do not have the same location; otherwise, false.
 </returns>

 <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
 <seealso cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
</member>
</members>
</doc>
```

Annex D. Non-normative references

ECMA-334:2005, C# *Programming language*.

Annex E. CLI naming guidelines

This annex is informative.

Information on this topic can be found at the following location:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp

End of informative text

Annex F. Future directions

This annex is informative.

This annex contains information about features that might be considered for a future revision of this Standard.

F.1 Expressions

F.1.1 Class member access

A named indexed property could be accessed like any other member of a class. [*Note:* As expected, an expression of the form `p->NamedIndexer[index]` is equivalent to `(*p).NamedIndexer[index]`. *end note*]

F.1.2 Type identification

Consider having a way for `typeid` on CLI class types produce a `std::type_info`.

F.1.3 Pointer type portability

The hardware architecture running the program determines the size of pointers. With the CLI, it is possible to use pointer types in programs that can run on multiple hardware architectures where pointer sizes are different. In order to support such programs, `sizeof` expressions on pointers would turn into a runtime expression instead of a compile time constant.

F.2 Statements

F.2.1 The checked and unchecked statements

Statements of the form `checked { … }` and `unchecked { … }` could be used to control the overflow-checking context for integral-type arithmetic operations and conversions.

F.3 Classes

F.3.1 Delegating constructors

Tutorial: When implementing a class, it is not unusual to have a number of constructors share some common code. For example, consider the case of the following point class:

```
class point {
   int x_;
   int y_;
   void commonCode();
public:
   point();
   point(int x, int y);
   point(const point& p);
   …
};
```

All three constructors need to initialize the two private members, `x_` and `y_`; they might also perform other actions, some of which they share, and some of which are unique. One approach is as follows:

```
point::point() : x_(0), y_(0) {
   commonCode();
   // custom code goes here
}
```

```
point::point(int x, int y) : x_(x), y_(y) {
   commonCode();
}
point::point(const point& p) : x_(p.x_), y_(p.y_) {
   commonCode();
   // custom code goes here
}
```

Certainly, the constructor with no parameters can be eliminated by adding default argument values to the constructor having two. However, that is not an entirely satisfactory approach for all classes. Specifically, it allows the two-argument constructor to be called with only the first argument, but not with only the second, which, philosophically, is asymmetric.

As shown above, a common approach to implementing such a family of constructors is to place their common code in a private member function, such as `commonCode`, and have each of them call that function.

C++/CLI could help solve this problem by providing ***delegating constructors***. Simply stated, prior to executing its body, a delegating constructor can call one of its sibling constructors as though it were a base constructor. That is, it delegates part of the Object's initialization to another constructor, gets control back, and then optionally performs other actions as well. Using this approach, the constructors shown earlier can be re-implemented as follows:

```
point::point() : point(0, 0) {
   // custom code goes here
}
point::point(int x, int y) : x_(x), y_(y) {
   // common code goes here
}
point::point(const point& p) : point(p.x_, p.y_) {
   // custom code goes here
}
```

Note how the *ctor-initializer* construct has been extended to accommodate a call to a sibling constructor, using the exact same approach as for a call to a base class constructor. The common code statements can now be part of the body of the second constructor, where they will be executed by calls to all three constructors. When the first and third constructors are called, they transfer control to the second. When that returns control to its caller, that caller's body is executed.

Any constructor can delegate to any of its siblings; however, a class shall have at least one non-delegating constructor (no diagnostic is required), and that constructor can still have a *ctor-initializer* that calls one or more base class constructors. A delegating constructor cannot also have a *ctor-initializer* that contains a comma-separated list of member initializers.

Specification: The definition of *ctor-initializer* is augmented to accommodate the addition of delegating constructors to C++/CLI; however, no change is necessary in the Standard C++ (§8.4) grammar.

Prior to executing its body, a constructor can call one of its sibling constructors to initialize members. That is, it delegates the object's initialization to another constructor, gets control back, and then optionally performs other actions as well. A constructor that delegates in this manner is called a ***delegating constructor***, and the constructor to which it delegates is called a ***target constructor***. A delegating constructor can also be a target constructor of some other delegating constructor. [*Example:*

```
class FullName {
   string firstName_;
   string middleName_;
   string lastName_;
public:
   FullName(string firstName, string middleName, string lastName);
   FullName(string firstName, string lastName);
   FullName(const FullName& name);
};
```

```
FullName::FullName(string firstName, string middleName, string lastName)
 : firstName_(firstName), middleName_(middleName), lastName_(lastName)
{
   …
}

// delegating copy constructor
FullName::FullName(const FullName& name)
 : FullName(name.firstName, name.middleName, name.lastName)
{
   …
}

// delegating constructor
FullName::FullName(string firstName, string lastName)
 : FullName(firstName, "", lastName)
{
   …
}
```

end example]

If a *mem-initializer-id* designates the class being defined, it shall be the only *mem-initializer*. The resulting *ctor-initializer* signifies that the constructor being defined is a delegating constructor.

A delegating constructor causes a constructor from the class itself to be invoked. The target constructor is selected by overload resolution and template argument deduction, as usual. If a delegating constructor definition includes a *ctor-initializer* that directly or indirectly invokes the constructor itself, the program is ill-formed; however, no diagnostic is required.

[*Example:* When using constructors that are templates, deduction works as usual:

```
class X {
   template<class T> X(T, T) : l_(first, last) { /* Common Init */ }
   list<int> l_;
public:
   X(vector<short>&);
};

X::X(vector<short>& v) : X(v.begin(), v.end()) { }
   // T is deduced as vector<short>::iterator
```

end example]

The object's lifetime begins when all construction is successfully completed. For the purposes of the C++ Standard (§3.8), "the constructor call has completed" means the originally invoked constructor call. [*Rationale:* Even if a target constructor completes, an outer delegating constructor can still throw an exception, and if so the caller did not get the object that was requested. The foregoing decision also preserves the Standard C++ rule that an exception emitted from a constructor means that the object's lifetime never began. *end rationale*]

F.3.2 Properties

Allowing properties in native classes.

Allowing the modifiers `abstract`, `new`, `override`, and `sealed` to be applied directly to a property as well as or instead of to one or more of its accessors.

F.3.3 Events

Allowing the modifiers `abstract`, `new`, `override`, and `sealed` to be applied directly to an event as well as or instead of to one or more of its accessors.

F.3.4 Unsupported CLS-recommended operators

Function Name in Assembly	C++ Operator Function Name

op_SignedRightShift	undefined
op_UnsignedRightShift	undefined
op_MemberSelection	undefined
op_PointerToMemberSelection	undefined

Regarding op_MemberSelection and op_PointerToMemberSelection, the C++ Standard only permits non-static member declarations of these operators.

F.3.5 Operators true and false

Add the ability to define operator true and operator false.

F.4 Generic types

Although the CLI permits the retrieval of a System::Type object that is associated with an open constructed generic type (§31.2.1), C++/CLI provides no syntax for doing this. However, such syntax might be considered in future.

F.5 Custom modifiers

F.5.1 IsPinned

This modopt type supports the use of the type pin_ptr as a parameter.

Description:

This type is used in the signature of any function. [*Example*:

```
public ref class X {
public:
   void F(pin_ptr<int> x) { … }
};
```

end example]

F.6 Attributes

Add the ability to chose unambiguously between two attributes called X and XAttribute.

End of informative text

Annex G. Portability issues

This annex is informative.

This annex collects some information about portability that appears in this Standard.

G.1 Undefined behavior

The committee that produced this standard did not intend to introduce any new undefined behavior.

G.2 Implementation-defined behavior

A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

1. Except for plain `char`, `signed char`, and `unsigned char`, the mapping of fundamental types to CLI types. (§12.1.1)
2. If the pre-defined macro `__cplusplus_cli` is the subject of a `#define` or a `#undef` preprocessing directive. (§11.1)

G.3 Unspecified behavior

The behavior is unspecified in the following circumstances:

1. Whether the replacement of an `__identifier` construct takes place before or after translation phase 4. (§9.1.1)
2. Whether white space generated by comments, documentation comments, and macro invocations is permitted in the position signified by the ⁘ symbol. (§9.1.2)
3. The semantics of any attribute target specifiers other than those described in this standard. (§29.2)
4. The interaction between the CLI library and the Standard C and C++ libraries (except for those requirements described elsewhere in this Standard). (§32)

End of informative text

Annex H. Index

This annex is informative.