Ecma/TC39-TG5/2004/25

# C++/CLI

## Language Specification

### Working Draft 1.5, Jun, 2004

### Public Review Document

Text highlighted like this indicates a placeholder for some future action. It might be a note from the editor to himself, or an indication of the actual or expected direction of some as-yet open issue.

Note: In the spirit of the "Working Draft, Standard for Programming Language C++", this is an early draft. It’s known to be incomplet and incorrekt, and it has lots of bad formatting.

Publication Time: 6/17/2004 11:44 PM

# Table of Contents

# Introduction

This International Standard is based on a submission from Microsoft. It describes a technology, called C++/CLI, that is a binding between the Standard C++ programming language and the ECMA and ISO/IEC Common Language Infrastructure (CLI) (§3). That submission was based on 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 ??.

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, had been formed in September 2000, to produce a standard for a library and execution environment called Common Language Infrastructure. An ISO/IEC version of that CLI standard (§3) has since been adopted. 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, 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 International Standard specifies requirements for implementations of the C++/CLI binding. The first such requirement is that they implement the binding, and so this International Standard also defines C++/CLI. Other requirements and relaxations of the first requirement appear at various places within this International 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 International 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.

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++*.

ISO/IEC 23271:2004, *Common Language Infrastructure (CLI)*, all Partitions.

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 instances of Objects 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 not defined by this specification. Essentially, an assembly is the output of the compiler. An assembly that has an entry point is called an application.

**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 a value class to type `System::Object`, in which an Object box is allocated and the value is copied into that box. (*See also* "unboxing".)

**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 an Object or class 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 allocated memory is automatically reclaimed on the CLI heap.

**gc-lvalue** — An expression that refers to an Object or subObject on the CLI heap.

**handle** — A handle is called an "Object reference" in the CLI specification. For any CLI 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 dynamic storage area (accessed by `new`) as defined in the C++ Standard (§18.4). (*See also* "heap, CLI".)

**IL** – Intermediate Language, the instruction set of the Virtual Execution System.

**instance** — An instance of a type; synonymous with "Object".

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

**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.

**Object** — An instance of a type; synonymous with "instance". (Uppercase-O Object is distinguished from the lowercase-o object defined in the C++ Standard.)

**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 and that address remains 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.

**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, boxed".

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

**type, class, interface** — A type that declares a set of virtual members that an implementing class must define. An interface class type binds to a CLI interface type.

**type, class, ref** — A type that can contain fields, function members, and nested types. Instances of a ref class type are allocated on the CLI heap. A ref class type binds to a CLI class 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 binds to a CLI value type.

**type, CLI** — An interface class, a ref class, or a value class.

**type, fundamental** — The arithmetic types as defined by the C++ Standard (§3.9.1), and that map to CLI value types. (These include `bool`, `char`, and `wchar_t`, but exclude enumerations.)

**type, handle** — Longhand for "handle".

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

**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 tracking reference is a kind of reference that has restrictions as to where it can be declared. For any type `T`, the declaration `T% r` declares a tracking reference `r` to type `T`. (*See also* "handle".)

**type, 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, value, simple** — The subset of value classes that can be embedded in a CLI type. The simple value classes include the fundamental types.

**unboxing** — An explicit conversion from type `System::Object` to any value class, from `V^` (the boxed form of a value class) to `V` (the value class), or from any interface class to any value class 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 IL 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 the 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. Additions to such text are indicated by underlining, and deletions are 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 relaxed:

"An 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 class handle, an enumeration, a reference to an enumeration, or an enumeration handle."

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, and they are described in §Annex D.

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.

# 6. Acronyms and abbreviations

**This clause is informative.**

The following acronyms and abbreviations are used throughout this 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

IEC — the International Electrotechnical Commission

IEEE — the Institute of Electrical and Electronics Engineers

ISO — the International Organization for Standardization

**End of informative text.**

# 7. General description

This Standard is intended to be used by 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, and semantics (clauses 9–32);
4. 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. Annexes provide additional information and summarize the information contained in this Standard.

Clauses 1–5, 7, and 9–32 form a normative part of this standard; and Foreword, Introduction, clauses 6 and 8, annexes, notes, examples, 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:* The following 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 chapters.

## 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\n
```

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

Look at the possibility of rewriting this sub-clause. C++ has many more class types, and a handle type can include all class types. Keep this placeholder until the type tree diagram has been added. [[#13]]

Value classes differ from handle types in that variables of the value classes 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 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 are of a value class (the type `int`) and each local variable of a value class has its own storage. In contrast, the assignment `ref2->Value = 123;` affects the 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 classes through enum and value class declarations. The example

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

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

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

public delegate void MyDelegate();
```

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

Types like `Color`, `Point`, and `IBase` above, which are not defined inside other types, can have a type visibility specifier of either `public` or `private`. The use of `public` in this context indicates that the type will be visible outside the assembly. Conversely, the `private` indicates that the type will not be visible outside the assembly. The default visibility for a type is `private`.

### 8.2.1 Fundamental types and the CLI

Each of the fundamental types is shorthand for a CLI-provided type. For example, the keyword `int` refers to the value class `System::Int32`. As a matter of style, use of the keyword is favored over use of the complete system type name.

The table below lists the fundamental types and their corresponding CLI-provided type: This mapping is still under discussion; it is by no means settled yet.[[#93]]

| Type | Description | CLI Value class |
|---|---|---|
| `bool` | Boolean type; a bool value is either true or false | `System::Boolean` |
| `char` | 8-bit signed/unsigned integral type | `System::SByte` or `System::Byte` (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` | 64-bit signed integral type | `System::Int64` |
| `unsigned long long` | 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` |

Add description for how fundamental types have the same member functions as those described in the CLI. [[Ed]]

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 28 significant digits

C++/CLI has no corresponding keyword for 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 Array types

An Array in C++/CLI 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 array element. The rank of an Array is also referred to as the ***dimensions*** of the Array. An Array with a rank of one is called a ***single-dimensional Array***, and an Array with a rank greater than one is called a ***multi-dimensional Array***.

Throughout this Standard, the term ***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.

Say more, especially w.r.t the template class array<element-type>. [[#23]]

### 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 as "3.".)

The example

```
int main() {
   int i = 123;
   Object^ o = i;                 // boxing
   int j = static_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 needs to be converted to a handle type, an Object ***box*** is allocated to hold the value, and the value is copied into the box. ***Unboxing*** is just the opposite. When an Object box is cast back to its original value class, 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 Objects, `int` values are simply 32-bit values. For programs that need `int` values to behave like Objects, this capability is available on demand. This ability to treat value classes as 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 classes like `int`.

### 8.2.5 Pointers, handles, and null

Standard C++ supports pointer types and null pointer constants. C++/CLI adds handle 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) has been extended 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 (zero is boxed and 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, C++/CLI defines handles (using the punctuator ^) and tracking references (using the punctuator %).

```
N* hn = new N;    // allocate on native heap
N& rn = *hn;      // 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, % is to ^ as & 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 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*** enables a many-to-one relationship: many arguments can be represented by a single parameter Array. Parameter arrays are a type safe alternative to parameter lists that end with an ellipsis.

A parameter array is declared with a leading `...` punctuator and an Array type. There can be only one parameter array for a given function, and it must always be the last parameter specified. The type of a

parameter array is always a single-dimensional Array type. A caller can either pass a single argument of this Array type, or any number of arguments of the element type of this 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 an 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 document 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) {…}
   };
}
```

[*Note:* 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)
```

*end note*]

## 8.4 Automatic memory management

The example

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

   property bool Empty {
      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) : Node(value, nullptr) {}
      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 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 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, 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 makes numerous additions and changes to 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 an Object of type `System::Type` that describes the CLI type `Int32`.
- The cast operators have been extended 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 have been extended to accommodate delegate addition and removal, respectively.
- Simple assignment has been extended to accommodate properties and events as the left operand.
- Compound assignment operators are synthesized from the corresponding binary operator. [[#56]]

## 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, a callable entity is just a member function. 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 is the address of the non-static member function within that ref class's type. The constructor for a delegate needs only one argument when it is bound to a static function, the argument is the address of the static member function.

The invocation lists of two compatible delegates can be combined via the `+=` operator, as shown. Also, 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 reference, 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 a constructor, or within a static constructor, and thereafter is an rvalue. This 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 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 a pure virtual member 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 scalar field-like access to an Object or class. 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 an 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 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() {
      move(0, 0);
   }

   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)
```

A ***default indexed property*** allows Array-like access directly on an instance. Whereas properties enable field-like access, default indexed properties enable Array-like access. [*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 can be 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) : Node(value, nullptr) {}
      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(Object^ value, int index) {
         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*]

```
Default indexed properties can just as easily be defined for native classes; for example:
public class IntVector {
public:
    property int default[int index] {       // default indexed property
        int get(int index) { … }
        void set(int index, int value) { … }
    }
…
};
int main() {
    IntVector iv(7, 5);   // define a 7-element vector with all values 5
    int i = iv[0];        // get element 0
    iv[1] = 55;                 // set element 1
    iv[3] -= 17;          // get and set element 3
    iv[5] *= 3;           // get and set element 5
}
```

### 8.8.5 Events

An ***event*** is a member that enables an Object or class 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).

An instance of a delegate type encapsulates one or more callable entities. For instance functions, a callable entity consists of an instance and a function on that instance. For static functions, a callable entity consists of just a 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.

In the example

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

public ref class Button {
public:
   event EventHandler^ Click;
   void Reset() {
      Click = nullptr;
   }
};
```

the `Button` class defines a `Click` event of type `EventHandler`. Inside the `Button` class, the `Click` member is exactly like a private field of type `EventHandler`. However, outside the `Button` class, the `Click` member is typically only used on the left-hand side of the `+=` and `−=` operators. 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, &Button1_Click);
   }
```

```
      void Disconnect() {
         Button1->Click -= gcnew EventHandler(this, &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.

For a trivial event declaration such as

```
event EventHandler^ Click;
```

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

An implementer 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 )    { Lock<Mutex> l(m); handler += e; }
      void remove( EventHandler^ e ) { Lock<Mutex> l(m); 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.

### 8.8.6 Static operators

Add examples for native and value classes. [[Ed]]

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 {
   int 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(length, 0) {}

   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

Introduce finalizers. [[#63]]

### 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, 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 shall 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 any class type. 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`.

```
struct A {
   virtual void f();
   virtual void h();
   virtual void j();
};
struct B {
   virtual void g();
   virtual void h();
};
struct D : A, B {
   virtual void f() override; // D::f overrides A::f
   virtual void g() override; // D::g overrides B::g
   virtual void h() override; // D::h overrides A::h and B::h
   virtual void j() new;      // D::j doesn't override A::j, it hides it
};
```

The use of `virtual` in `D::f`, `D::g`, and `D::h`, is mandatory; however, that in `D::j` is not.

***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.

```
struct A {
   virtual void j();
   virtual void m();
};
struct B {
   virtual void k();
};
struct D : A, B {
   virtual void x() = A::j;         // D::x overrides A::j
   virtual void y() = A::m, B::k;   // D::y overrides A::m and B::k
};
struct P {
   void f();
private:
   virtual void h();
   virtual void j();
};
struct Q : P {
   virtual void f() = P::f;   // error, P::f is not overridable
   virtual void h();          // P::h not visible, but ok
};
```

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

Explicit and named overriding can be combined, as follows:

```
struct A {
   virtual void f();
   virtual void g();
};
struct D : A {
   virtual void f() override = A::g;         // D::f overrides A::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.

```
struct B {
   virtual void f();
};
struct D : B {
   virtual void f() override = B::f;   // Error: B::f 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 class R : T {
public:
   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 structs. Key to these data structures is that they have few fields, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

The simple types provided by C++/CLI, such as `int`, `double`, and `bool`, are, in fact, all value classes. Just as these predefined types are value classes, it is also possible to use value classes and operator overloading to implement new “primitive” types in this specification.

```
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];
   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 class 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,
      public IDataBound {
public:
   void Paint() {…}
   void Bind(Binder^ b) {…}
};
```

the class `EditBox` derives from the 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 override 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:
   void Paint() = IControl::Paint {…}
   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 promotion conversions.

## 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 a `#using` directive, 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 evolving 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

C++/CLI 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 declaration 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 declaration 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 declaration `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 declaration 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 declarations can have any number of type parameters. Suppose we created a simple `Dictionary` generic class declaration 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(ElementType value, KeyType key) { … }
   }
};
```

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 declaration. 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(x) < 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(x) < 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(x) < 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(x) < 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 declaration, 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 ItemType>
void PushMultiple(Stack<ItemType>^ 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, 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>(s, 1, 2, 3, 4);
```

The call to this function supplies the desired `ItemType` as a type argument 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);
```

Based on the rules for type deduction in templates, it seems surprising that you can match array<ItemType>^ with an argument of type int. Here is a standard C++ example intended to illustrate the issue:

```
template <class ItemType> struct Stack {};
template <class ItemType> struct Array {
    Array(ItemType);
};

template <class ItemType>
void PushMultiple(Stack<ItemType>, Array<ItemType>);
int main() {
     Stack<int> s;
     PushMultiple(s, 1);  // deduction fails
     PushMultiple<int>(s, 1);
}
```

Are the rules for generic different in this area?

[There seems to be information related to this in 30.3.2. See that subclause for further comments on this issue.][[#125]]

**End of informative text.**

# 9. Lexical structure

A number of issues are not yet discussed here. Much of this clause is yet to be added. [[#24]]

## 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; none-the-less, they are identifiers. *end note*] The identifiers that carry special meaning in certain contexts are:

```
abstract    delegate    event       finally
generic     in          initonly    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.

When the token `generic` is found, it has special meaning if and only if it is not preceded by the token `::` 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 extended as follows, to make *nested-name-specifier* optional in the first of the two applicable productions:

*elaborated-type-specifier:*

...
`typename` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
`typename` `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id*
...

The C++ standard (§14.6/3) is amended, as follows:

"A ~~*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 parenthesis, 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.

### 9.1.2 Keywords

The following keywords are added to those in the C++ Standard (§2.11):

```
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, including comments and new-lines (but excluding documentation comments and newlines in macros), 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 extended 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. If a character sequence that matches the syntax of a universal-character-name is produced by token concatenation (16.3.3), the behavior is undefined. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

In some places in the grammar, specific 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) has been extended as follows:

*literal:*
…
*null-literal*

#### 9.1.3.1 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.

### 9.1.4 Operators and punctuators

It has been agreed that >> will be handled apprpriately to allow constructs such as List<List<int>> to be handled correctly. If a < for a template, for example, is seen, and >> that is not inside parentheses, that >> will always be considered to be the closing delimiter of two < symbols, and results in an error if there are not two such corresponding < symbols. [[Ed.]]

# 10. Basic concepts

Much of this clause is yet to be added, include application entry point, assembly boundaries, etc. [[#25]]

#using subclause: When importing functions from an assembly, functions with these names shall be renamed with the appropriate C++ identifier for the conversion function. If such a function does not make sense as a conversion function (for example, it takes three arguments), the function name is not changed to the internal conversion function name, and thus the function is callable by the name it has in the assembly. [[#95]]

## 10.1 Members

### 10.1.1 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 correspond directly to the members of the value class type aliased by the fundamental type, as follows: This mapping is still under discussion; it is by no means settled yet.[[#93]]

- The members of `signed char` are the members of the `System::SByte` value class.
- The members of `unsigned char` are the members of the `System::Byte` value class.
- If a plain `char` is signed, the members of `char` are the members of the `System::SByte` value class; otherwise, they are the members of the `System::Byte` value class.
- The members of `short int` are the members of the `System::Int16` value class.
- The members of `unsigned short` are the members of the `System::UInt16` value class.
- The members of `int` are the members of the `System::Int32` value class.
- The members of `unsigned int` are the members of the `System::UInt32` value class.
- The members of `long long` are the members of the `System::Int64` value class.
- The members of `unsigned long long` are the members of the `System::UInt64` value class.
- The members of `wchar_t` are the members of the `System::Char` value class.
- The members of `float` are the members of the `System::Single` value class.
- The members of `double` are the members of the `System::Double` value class.
- The members of `long double` are the members of the `System::Double` value class.
- The members of `bool` are the members of the `System::Boolean` value class.

### 10.1.2 Delegate members

The members of a delegate are the members inherited from class `System::Delegate`, in addition to the members added by the C++ compiler. [*Note:* The compiler needs to add typedef members to the class so that template code can use the return type or the parameter types. *end note*]

## 10.2 Member access

### 10.2.1 Declared accessibility

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

*access-specifier:*

```
...
public private
private public
protected public
public protected
private protected
protected private
public public
protected protected
private private
```

It is expected that "public private" (and "private public") will be replaced by "internal", and that those access-specifiers containing the same name twice will simply revert to a single occurance of that name. [[Ed.]]

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

A member of a class can be

- `private` or `private private`; that is, its name can be used only by members and friends of the class in which it is declared.
- `protected` or `protected 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.
- `public` or `public public`; that is, its name can be used anywhere without access restriction.
- `public private` or `private public`; 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. .

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.

An overriding name is allowed to have a different accessibility than the name it is overriding. Clarify the ordering definition. [[#26]] 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 the same or less access than A within the assembly and outside the assembly. A has wider access than B if A permits the same or more access than A within the assembly and outside the assembly. Narrowing and widening of accessibilities implies a partial ordering of accessibilities. For example, `protected` is wider than `private`, `protected` is wider than `protected`, `protected` is narrower than `public`, `protected` is narrower than `protected`, `protected private` is narrower than `public protected`, and no ordering exists between `public private` and `protected`. [*Note:* In general, widening and narrowing accessibility is not CLS compliant. *end note*]

# 11. Preprocessor

## 11.1 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 200406L 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 undefined.

# 12. Types

Add a picture of a type tree. [[#13]]

The C++ Standard (§3.9/10) definition for ***scalar types*** has been extended, as follows:

"Arithmetic types (3.9.1), enumeration types, handles, 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* has been extended, as follows:

*type-specifier:*
…
*delegate-definition*

## 12.1 Fundamental types

Standard C++ (§3.9.1) is augmented by the following: This mapping is still under discussion; it is by no means settled yet.[[#93]]

- 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.
- There are five ***signed integer types***: "`signed char`", "`short int`", "`int`", "`long int`", and "`long long`"
- For each of the signed integer types, there exists a corresponding (but different) unsigned integer type: "`unsigned char`", "`unsigned short int`", "`unsigned int`", "`unsigned long int`", and "`unsigned long long`"
- An object of type `short int` shall have exactly 16 bits.
- An object of type `int` shall have exactly 32 bits.
- An object of type `long int` shall have exactly 32 bits.
- An object of type `long long` shall have exactly 64 bits.
- The value of an object having a signed integer type shall be stored using twos-complement representation.
- An object of type `wchar_t` shall be unsigned and have exactly 16 bits.
- An object of type `float` is represented using the 32-bit single-precision IEC 60559 format.
- An object of type `double` is represented using the 64-bit double-precision IEC 60559 format.
- An object of type `long double` is represented using the 64-bit double-precision IEC 60559 format.
- An object of type `bool` shall have exactly 8 bits.

The type long long will be defined by pointing to the paper WG21 N1565.[[#126]]

## 12.2 Class types

### 12.2.1 Native classes

### 12.2.2 Value classes

Is there more to say? What about boxing? [[Ed]]

All value class types implicitly inherit from the class `System::ValueType`, which, in turn, inherits from class `System::Object`. [*Note:* `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. *end note*]

#### 12.2.2.1 Simple value classes

Is this the place to describe the mapping of fundamental types to CLI types? [[Ed]]

#### 12.2.2.2 Enum classes

### 12.2.3 Ref classes

A ref class defines a data structure that contains fields, function members (functions, properties, events, operators, instance constructors, destructors, and static constructors), and nested types. Ref classes support inheritance. Instances of ref classes are created using *new-expression*s (§15.4.6.1).

Ref classes are described in §20.

### 12.2.4 Interface classes

An interface defines a contract. A ref or value class that implements an interface must 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 §24.

### 12.2.5 Delegate types

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

Delegate types are described in §26.

### 12.2.6 Arrays

## 12.3 Declarator types

The WG21 WP says at the end of 8.3.5p3 "The resulting list of transformed parameter types and the presence or absence of the ellipsis is the function's parameter-type-list." Since we are using the term ”parameter-type-list", we need to define it in this clause somewhere. [[Ed]]

### 12.3.1 Raw types

### 12.3.2 Pointer types

It is possible to declare a pointer to a function that takes a parameter array (§18.3.6). [*Example:*

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

*end example*]

A native pointer cannot point to an Object on the CLI heap unless that Object has been pinned (§12.3.7).

### 12.3.3 Handle types

Need to add text to indicate the circumstances under which the modreq IsBoxed shall be emitted (i.e., passing a handle to a value type). Point to that modreq's spec.[[#127]]

For any CLI 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. [*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 is `nullptr`.

Objects of CLI 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 point 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 (§??), 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 Object to which it refers gets 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 Object.

[*Example:*

```
R^ r4 = new 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 Object compare equal, even if that Object is moved by the garbage collector.

A handle can have any storage duration.

### 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. [*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*]

Like an ordinary 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. [*Note:* This limitation directly reflects that of the CLI, because tracking references are in general implemented in terms of CLI byrefs. This limitation is not inherent in this language design, and can be removed on CLI platforms that support byrefs that can exist in non-stack locations. *end note*]

### 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_ptr` is a pointer that is updated in this manner.

We need a grammar for this. [[#108]]

The compiler will need to emit a modopt to distinguish interior_ptr<T> from tracking reference to T (T%) in the metatada.[[#28]] Need to add text to indicate the circumstances under which the modopt IsExplicitlyDereferenced shall be emitted (i.e., interior_ptr as a parameter). Point to that modopt's spec.

#### 12.3.6.1 Definitions

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

An interior pointer shall not be a subObject.

The default initial value for an interior pointer not having an explicit initial value, shall be `nullptr`.

[*Note:* An interior pointer to a value class can be implemented as a CLI byref. However, a byref can't refer to a whole Object, so an interior pointer to a ref class can be implemented using an Object reference (just like a handle is implemented); this common implementation need not affect the programmer, who still sees distinct semantics for `interior_ptr<R>` and `R^`. *end note*]

#### 12.3.6.2 Target type restrictions

An interior pointer shall not point to a ref class Object. (However, such a pointer is permitted to point to a handle to a ref class Object.) Other target types are permitted. We need to say which types. For example, what about pointers to functions? [[#29]] [*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;   // OK
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*]

Cover the dangers of pointer arithmetic and interior_ptrs. [[#109]]

#### 12.3.6.4 Conversion rules

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 `nullptr` is)

[*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*]

#### 12.3.6.5 Data access

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

- Operator `->` is used to access a member of an 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*`, as described in §12.3.6.4.

[*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*]

#### 12.3.6.6 The this pointer

In the body of a non-static member-function of a value class `V`, `this` is an expression of type `interior_ptr<V>`, whose value is the address of the 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

Need to add text to indicate the circumstances under which the modopt IsPinned shall be emitted (i.e., pin_ptr as a parameter). Point to that modopt's spec.[[#129]]

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*** is one 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.

#### 12.3.7.1 Definitions

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

We need a grammar for this. [[#110]]

[*Note:* As a pinning pointer is an interior pointer, the default initial value for a pinning pointer not having an explicit initial value, 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).

#### 12.3.7.4 Conversion rules

The following conversion rules apply to interior 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 `nullptr` is). [*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*]

### 12.3.7.5 Data access

With two exceptions, pinning pointers follow the same data access semantic as interior pointers (§12.3.6.5). Since a pinning pointer points to an unmovable Object inside the CLI heap, a `pin_ptr<T>` can be converted to `T*` (§12.3.7.4). 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*]

### 12.3.7.6 Duration of pinning

As soon as a pinning pointer is initialized or assigned the address of an Object, that Object is guaranteed to remain at its location on the CLI heap. If the pinning pointer is then made to point to another Object, that Object is guaranteed to remain at its location on the CLI heap, and the Object previously pointed to is no longer considered pinned, allowing the garbage collector to move it. 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 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.

[*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*]

## 12.4 Top-level type visibility

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

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

The `public` *top-level-type-visibility* specifier indicates that the non-nested class, interface, delegate, or enum will be visible outside the assembly. Conversely, the `private` *top-level-type-visibility* specifier indicates that the class, interface, delegate, or enum will not be visible outside the assembly. However, private types are visible within the same 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-type-visibility* modifier on a nested type definition causes the program to be ill-formed.

# 13. Variables

To be added.[[#32]]

# 14. Conversions

## 14.1 Standard conversions

The standard conversions in the C++ standard apply to C++/CLI. The following standard conversions are added:

### 14.1.1 Handle conversions

A handle conversion is similar to a pointer conversion as defined in the C++ Standard (§4.10). A handle conversion has conversion rank.

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`. If `B` is an inaccessible or ambiguous base class of `D`, a program that necessitates this conversion is ill-formed. The result of the conversion is a handle to the base class sub-Object of the derived class 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 Arrays) is the same. Such a conversion is better than Separate the list of conversions from the order of preference (such as how Standard C++ separates Standard Conversions from overload resolution). a conversion from type `array<S^, n>` to `System::Array^`.

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 Object. Two null values of the same handle type shall compare equal.

### 14.1.2 Pointer conversions

The definition of ***null pointer constant*** in the C++ Standard (§4.10/1) has been extended, 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*]

Need to say more here. Possibly move "Interior pointer conversion rules" (§12.3.6.4) and "Pinning pointer conversion rules" (§12.3.7.4) here. [[Ed]]

### 14.1.3 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."

## 14.2 Implicit conversions

The C++ Standard (§4.12) text that describes Boolean conversions has been extended, as follows:

"An rvalue of arithmetic, enumeration, pointer, pointer to member type, or handle can be converted to an rvalue of type `bool`. A zero value, null pointer value, null member pointer value, or null value is converted to `false`; any other value is converted to `true`."

### 14.2.1 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.

### 14.2.2 User-defined implicit conversions

## 14.3 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.

## 14.4 Boxing conversions

The boxing conversion applies only to value classes (including the simple value classes). The boxing conversion cannot be rewritten by the user and is reserved to the implementation.

The boxing conversion is modeled as a preferred UDC. The text of this section should be revised to address concerns from the updated conversion proposal. [[#34]]

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 revised, as follows:

We should start off the conversions clause with "Conversion Sequences", which would cover this adjustment to the C++ Standard. That makes Boxing conversions shorter and prevents us from introducing parameter array conversions in a sub-clause where it doesn't belong. [[#34]]

"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)."

The boxing conversion for a value class `V` is an implicit conversion from `V` to `V^`. As stated above, a standard conversion is permitted to follow a boxing conversion, and thus a handle conversion is able to convert `V^` to `System::Object^` or a handle to an interface that `V` implements. The conversion occurs as follows:

The compiler selects the boxing conversion and emits the BOX instruction as specified in the CLI Standard, Partition III, §4.1. This causes a runtime bitwise copy of the value class instance to an Object on the CLI heap.

All value classes must be copyable. That is, a value class shall not have a non-public default constructor.

Ref classes have an explicit conversion from `R` to `R^`. (This is described later in §??.)

## 14.5 User-defined conversions

### 14.5.1 Constructors

All constructors in ref and value classes are explicit (C++ Standard, §12.3.1). Using the `explicit` keyword on a constructor in a ref class or value class is permitted, but it is redundant.

The meaning of an explicit constructor is unchanged from Standard C++. [*Note:* That is, an explicit constructor is permitted in direct-initialization syntax (C++ Standard, §8.5) and casts (C++ Standard, §5.2.9, §5.4). *end note*]

Further changes are needed to effectuate the CLI convention that constructors are never used for conversions, whether explicit or implicit. Making constructors of ref and value classes explicit eliminates them from consideration for implicit conversions, but additional changes to the overload resolution rules are needed to indicate that such constructors should be considered for casts of the form X(c) (which are viewed as creating an object) but not for casts of other forms, e.g., (X)(c) or static_cast<X>(c) (which are viewed as conversions). The C++ standard treats those two cases as equivalent direct-initializations.[[#105]]

### 14.5.2 Explicit conversion functions

C++/CLI allows the `explicit` keyword on conversion functions. Thus, C++ Standard, §7.1.2 is changed, 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."

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***.

An explicit conversion function, like an explicit constructor, 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.

Add an example. [[Ed]]

### 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&`, 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 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. This is similar to the tiebreaker rules for template-functions and non-template functions. It would be useful to reference those somehow. [[Ed]]

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 an Array of length `n` and initializes it with the converted values. Then the function call is made with the constructed parameter array.

## 14.7 Compiler-defined explicit conversions

### 14.7.1 Unboxing conversions

The unboxing conversion allows a conversion to an unboxed value class directly from a handle to one of the following:

- `System::Object`
- `System::ValueType`
- an interface that the value class implements
- the value class itself

The conversion from the boxed form of a value class (`V^`) to the value class (`V`) can be done using a dereference (i.e., `operator*`). It can also be done by any cast notation that invokes user-defined conversions.

The unboxing conversion can be done with any cast notation that invokes user-defined conversions.

## 14.8 Naming conventions

Conversion functions shall conform to a particular naming convention. (The names required of conversion functions are given by the CLS guidelines.) While all conversion functions have the CLS required name, not all conversion functions are CLS-conversion functions.

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()`. The identifier used for the operator function in an assembly shall have the CLS name as specified in §14.8.1 and §14.8.2.

A conversion function inside a native class shall have the names used in §14.8.1 and §14.8.2 prefixed with `<` and suffixed with `>`. Otherwise, the name specified in these subclauses is unchanged. A C++ program shall not declare nor define a function within a CLI type using one of the CLS names referred to herein.

A program shall not refer to the CLS-compliant name given to the conversion function.

All conversion functions, regardless of whether they are CLS-compliant functions or not shall be marked as ***SpecialName*** functions in the metadata.

### 14.8.1 CLS-compliant conversion functions

A conversion function is CLS-compliant when 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 the above criteria are not met, the conversion function is C++-dependent. Table 14-1 lists the name to give to the function used to represent the operator function in an assembly.

**Table 14-1: CLS Conversion Functions**

| Function Name in Assembly | C++ Conversion Function |
|---|---|
| `T op_Implicit(S)` | `operator T(S)` |
| `T op_Explicit(S)` | `explicit operator T(S)` |

The operators `op_Implicit` and `op_Explicit` are permitted to be overloaded on their return type.

### 14.8.2 C++-dependent conversion functions

If a conversion function does not match the criteria for CLS compliance, as listed in §14.8, the conversion function is C++-dependent. The names in Table 14-1 are also used for C++-dependent conversion functions in an assembly.

Both `op_Implicit` and `op_Explicit` are allowed to be overloaded on their return type.

Converting constructors are emitted as constructors, never as converting functions. (Constructors in CLI classes are always explicit.)

# 15. Expressions

## 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, `T` indicates an expression classified as a type, `F` is the simple name of a function, 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::get(x, y)` |
| | `e[x, y] = value` | `E::set(x, y, value)` |

The rewrite rules for e[x] (default indexed accesses) are different where there is only one index. This is because there is a potential ambiguity with the C++ operator[]. Is this mentioned elsewhere? [[#35]]

*end note*]

## 15.2 Primary expressions

To accommodate the addition of properties, the "Primary expressions" subclause of the C++ Standard (§5.1) has been extended, 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 Arrays (which are accessed using subscript-like expressions), the C++ Standard grammar (§5.2) for postfix-expression has been extended, as follows:

*postfix-expression:*
...
*postfix-expression* `[` *expression* `]`
*indexed-access*

Indexed access is described in §15.3.2.

### 15.3.1 Subscripting

Given a class instance `x`, of a type having a default indexed property and `operator[]`, an expression of the form `x[i]` is ambiguous. In such cases, the `operator[]` function or default indexed property accessor function must be called directly, as appropriate. If a derived class defines only one of `operator[]` or a default indexed property, lookup will use that function rather than making the program ambiguous.

### 15.3.2 Indexed access

An *indexed-access* consists of an *indexed-designator*, followed by a "`[`" token, followed by an *expression-list*, followed by a "`]`" token. The *expression-list* consists of one or more *expression*s, separated by commas.

*indexed-access:*
*indexed-designator* `[` *expression-list* `]`

*indexed-designator* shall designate an instance that has one or more default indexed properties that are applicable with respect to the *expression-list* of the *indexed-access*.

An *indexed-access* is interpreted as follows: Each default indexed property with only one indexing parameter has an associated `operator[]` synthesized. For the property `property int default[int]`, the synthesized "`operator[](int)`" is created. Overload resolution for the appropriate `operator[]` is done for *indexed-access* expressions where the expression list is not comma-separated. If a class has two `operator[]` operators with the same signature, the expression is ambiguous and the program is ill-formed. Otherwise, the rewrite rules for properties and events are used for *indexed-access* expressions.

Need to consider how these expressions are interpreted in templates. [[#111]]

Commas in *expression-list* are treated as a special case—they are considered punctuators. However, if an expression in that list is enclosed in parentheses, any commas inside that expression are interpreted as operators (and behave as described in §5.18/2 of the C++ Standard).

```
struct S {
   property int default[int index] { … }              // indexed property
1
   property int default[string idx1, int idx2] { … }  // indexed property
2
};

void f(S& s, string& x, int j) {
   s[x,j]         = 42; // ok, uses indexed property 2
   s[1,j]         = 42; // error (tries to use indexed property 2,
                        // but there is a type mismatch;
                        // no comma operator is used)
   s[(1,j)]    = 42; // ok, uses indexed property 1 with j as the
argument
   s[(1,x),j]  = 42; // ok, uses indexed property 2
}
```

[*Note:* Given a class instance `x`, of a type having a default indexed property and `operator[]`, an expression of the form `x[i]` can be ambiguous. In such cases, the `operator[]` function or default indexed property accessor function must be called directly, as appropriate. *end note*]

### 15.3.3 Function call

Add text to indicate the circumstances under which the following type modifiers shall be emitted, and point to each modifier's definition:

- IsBoxed i.e., passing a handle to a value type).

- IsByValue (i.e., ref class type passed by value).
- IsConst (i.e., pointer or reference to a const-qualified type).
- IsExplicitlyDereferenced (i.e., interior_ptr as a parameter).
- IsImplicitlyDereferenced (i.e., parameter is a reference).
- IsLong (i.e., long/unsigned long/long double parameters).
- IsExplicitlyDereferenced (i.e., pin_ptr as a parameter).
- IsSignUnspecifiedByte (i.e., plain char's sigedness).
- IsUdtReturn (i.e., ref class type returned by value).
- IsVolatile (i.e., pointer or reference to a volatile-qualified type).[[#131]]

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 (§26). As such, the postfix expression can be a delegate type, in which case, the whole expression is a delegate invocation (§26.3), and the argument list is passed to each function encapsulated by the delegate.

### 15.3.4 Explicit type conversion (functional notation)

### 15.3.5 Pseudo destructor call

### 15.3.6 Class member access

A named indexed property is 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*]
If a program attempts to access a default indexed property via a pointer to an Object having that default indexed property, and the arrow operator, that program is ill-formed. [*Note:* Although `p->[index]` is ill-formed, the expression `(*p)[index]` is permitted. *end note*]

### 15.3.7 Increment and decrement

### 15.3.8 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 neither a handle nor a pointer, it is possible for dynamic cast expressions to invoke an unboxing conversion. If `T` is a value class, and `e` has type `T^` or a type `U^` (where there is a handle conversion from `T^` to `U^`), the dynamic cast invokes the UNBOX instruction from the CLI Standard, Partition III. If `T` is a `V%` for a value class `V`, and `e` has type `V^` or a type `U^` (where there is a handle conversion from `V^` to `U^`), the dynamic cast invokes the UNBOX instruction as well. If the unboxed type is not of type `T`, then an exception of type `System::InvalidCastException` is thrown. cv-qualification needs to be considered. [[#36]]

Otherwise, if `T` is a native reference to a value class, and `e` has type `U^`, the program is ill-formed. [*Rationale:* This can open a gc hole in the program as native references do not track what they refer to during garbage collection. *end rationale*]

Otherwise, if `T` is `V^` (where `V` is a value class) or `U^` (where there is a handle conversion from `V^` to `U^`), and `e` has a type `V` or reference to `V`, then the expression invokes a boxing conversion sequence.

Otherwise, if `T` is a handle type, `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, 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 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 a run-time check is applied to the cast, and `T` is a handle or reference to a CLI type, the run-time check is performed using the `ISINST` CIL instruction from the CLI Standard, Partition III, §4.6.

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.

### 15.3.9 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.

The C++ Standard grammar production for *unary-expression* (§5.3 and §A.4) is extended with a new production as follows:

*unary-expression:*
`...`
*typeid-expression*

*typeid-expression:*
*elaborated-type-specifier* `:: typeid`

In the C++ standard (§14.6.2.2/4), the "Expressions of the following forms" list is extended to include *typeid-expression*.

The result of a *typeid-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 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-expression* shall be a raw type or a pointer to a raw type.
Check if long::typeid and char::typeid are allowed (and if so, what do they mean). [[#112]]

Add a note that discourages the practice of using the result of T::typeid to guard static members with a lock. [[Ed]]

The *typeid-expression* provides convenient syntactic access to the functionality of the `System::Type::GetType()` library function. Whereas `GetType()` must be called on an Object of the given type, `::typeid` can be applied to a type directly, and consequently does not require an 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.Single[]
System.Void
System.Object
System.String ToString(System.IFormatProvider)
System.TypeCode GetTypeCode()
System.Object Clone()
…
System.String IsInterned(System.String)
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 an 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>` (§30.1.2). *end example*]

It might be useful to add an example showing the use of the ::typeid-form with a custom attribute.

What about handles and tracking references? We still need to make sure we have a design for standard typeid (that returns std::type_info) in addition to the new ::typeid (that returns System::Type). [[#38]]

### 15.3.10 Static cast

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

Unboxing and boxing are described as preferred user-defined conversions. Nothing important about these needs to be mentioned in static cast, but those UDCs are not completely specified yet.[[#132]]

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.

The cast expression discussed in the C++ Standard (§5.2.9/3) is allowed also 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 (§14.1.1), 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. This can be unverifiable and might cause a gc hole.[[#133]]

### 15.3.11 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 an unboxing conversion or a boxing conversion sequence.

### 15.3.12 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 will never invoke an unboxing conversion or a boxing conversion sequence.

### 15.3.13 Safe cast

Safe cast performs the optimal cast for CLI frameworks. The name `safe_cast` is located within the `cli` namespace. 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`, 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 type of the operand and the target type shall be a value class, a handle to a value class, a handle to a ref class, or a handle to an interface class. Otherwise, the expression is ill-formed.

Include the specification for safe_cast from the revised casting proposal. [[#39]]

## 15.4 Unary expressions

### 15.4.1 Unary operators

#### 15.4.1.1 Unary &

Since a discussion of lvalue, rvalue, and gc-lvalue has now been included, the above statement is generalized by saying that the application of & to an rvalue or a gclvalue is ill-formed. (Is this still true?) [[#40]]

When applied to an lvalue of type T, & yields a `T*` (see Standard C++ §??). 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 a literal field, or to a property, or to an initonly field outside of the class's constructor, is ill-formed.

#### 15.4.1.2 Unary *

The C++ Standard (§5.3.1/1) has been extended 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 type, 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`.

#### 15.4.1.3 Unary %

When applied to an lvalue of type `T` or a gc-lvalue of type `T`, % yields a `T^`. [*Example:*

```
ref class R { };
void f(System::Object^);
R r;
f(%r);   // ok
```

*end example*]

This operator results in a boxing operation. [*Note:* All handles to the same Object compare equal. For value classes, because % is a boxing operation, multiple applications of % results in a handles that do not compare equal. *end note*]

#### 15.4.1.4 Unary ^

No such operator exists; should it? The only major asymmetry between %/^ and &/* is that unary * is used to dereference both * and ^, which allows for the writing of templates that can deal with both pointer and handle types using a common syntax; however, there is no unary ^. People new to the syntax often expect to dereference a ^ using a unary ^. Should unary ^ be allowed as a synonym for unary *? Doing so might introduce needless redundancy by having two unary operators with identical semantics. We might also be closing a door if we later discover a valid distinct meaning for unary ^ vs. unary *—we can't think of any meaning but the single "dereference" meaning, but maybe we're just not imaginative enough.)[[Ed.]]

### 15.4.2 Increment and decrement

### 15.4.3 Sizeof

The mapping of C++/CLI types to fundamental types is still under discussion; it is by no means settled yet, so the sizeof guarantees below may change or be removed.[[#93]]The C++ Standard (§5.3.3/1) has been extended, 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)` are implementation-defined.~~ `sizeof(short)` is 2, `sizeof(int)` is 4, `sizeof(long)` is 4, `sizeof(long long)` is 8, `sizeof(float)` is 4, `sizeof(double)` is 8, `sizeof(long double)` is 8, `sizeof(wchar_t)` is 2, `sizeof(bool)` is 1. *end note*]"

The following paragraph is inserted after C++ Standard (§5.3.3/2):

> "When applied to a value class type, the result is not a compile-time constant expression."

### 15.4.4 New

A program is ill-formed if it attempts to allocate memory using `new` for an Object of CLI type other than a simple value class.

### 15.4.5 Delete

The C++ Standard (§5.3.5/1) has been extended 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 not, the behavior is undefined."
>
> "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` cannot be used on a handle type.

### 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 type using `gcnew`.

#### 15.4.6.1 gcnew Object creation expressions

In the C++ Standard (§5.3.4), a *new-expression* is used to allocate memory for an Object at runtime. This grammar has been extended to accommodate the addition of the `gcnew` operator, as follows:

*new-expression:*
…
`gcnew` *new-type-id* *new-initializer*$_{opt}$
`gcnew` `(` *type-id* `)` *new-initializer*$_{opt}$

Add the array case to this grammar. [[#42]]

The type of the Object being allocated shall not be an abstract class type. The type shall not be incomplete. [*Note:* The `gcnew` operator applied to a value class creates a boxed value class. *end note*]

#### 15.4.6.2 Array creation expressions

Does new-initializer need to be changed? [[#114]]

## 15.5 Explicit type conversion (cast notation)

The rules in the C++ Standard (§5.4/5) have been extended 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*]

Provide background on the expected behavior and rationale. (Get this from the updated casting proposal.) [[Ed]]

## 15.6 Pointer-to-member operators

## 15.7 Multiplicative operators

## 15.8 Additive operators

### 15.8.1 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` on both arguments, and casting the result to `D^`. [*Note:* For examples of delegate combination, see §15.8.2 and §26.3. Since `System::Delegate` is not a delegate type, `operator+` is not defined for it. *end note*]

### 15.8.2 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. *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*]

## 15.9 Shift operators

## 15.10 Relational operators

## 15.11 Equality operators

### 15.11.1 Ref class equality operators

Add support for handle equality comparison, and handle ==/!= nullptr, and vice versa. [[#43]]

### 15.11.2 Delegate equality operators

Every delegate type 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`.

## 15.12 Bitwise AND operator

## 15.13 Bitwise exclusive OR operator

## 15.14 Bitwise inclusive OR operator

## 15.15 Logical AND operator

## 15.16 Logical OR operator

## 15.17 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 changed 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."

## 15.18 Assignment operators

Add words here to discuss assignment for properties and events from the point of view of the rewrite rules. [[#44]]

The left operand of an assignment shall be an lvalue or a gclvalue.

## 15.19 Comma operator

## 15.20 Constant expressions

The C++ Standard (§5.19/2) provides a list of "Other expressions [that] are considered *constant-expressions* only for the purpose of non-local static object initialization." That list has been extended 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 following is inserted in the C++ Standard, after §5.19/3:

5 "A ***literal constant expression*** includes ***arithmetic constant expression***, string literals of type `System::String`, and the null value constant `nullptr`."

Investigate whether string literals include compile-time expressions, such as string concatenation. [[#115]]

# 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 has been extended to include *foreach-statement*.

*iteration-statement:*
…
*foreach-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.

*foreach-statement:*
`for each (` *type ??-declaratoropt identifier* `in` *expression* `)` *statement*

The *type*, *declarator*, and *identifier* of a `for each` statement declare the ***iteration variable*** of the statement. The iteration variable corresponds to a local variable with a scope that extends over the substatement. During execution of a `for each` statement, the iteration variable represents the collection element for which an iteration is currently being performed. The program is ill-formed if the substatement attempts to assign to the iteration variable or to pass the iteration variable by reference.

The type of *expression* shall be a collection type (as defined below), and an explicit conversion (§??) must exist from the element type of the collection to the type of the iteration variable. If *expression* has the value `nullptr`, a `System::NullReferenceException` is thrown.

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

- `C` contains a `public` instance function with the signature `GetEnumerator()`, that returns a *struct-type*, *class-type*, or *interface-type*, which is called `E` in the following two points.
- `E` contains a `public` instance function with the signature `MoveNext()` and the return type `bool`.
- `E` contains a `public` instance property named `Current` that permits reading the current value. The type of this property is said to be the ***element type*** of the collection type.

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 (§23.1.1) is a collection type, and since all Array types derive from `System::Array`, any Array type expression is permitted in a `for each` statement. For single-dimensional Arrays, the `for each` statement enumerators traverses the Array elements in increasing order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional 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 is executed as follows:

- The collection expression is evaluated to produce an instance of the collection type. This instance is referred to as `c` in the following.
- An enumerator instance is obtained by evaluating the function invocation `c.GetEnumerator()`. The returned enumerator is stored in a temporary local variable, in the following referred to as `e`. It is not possible for the statement to access this temporary variable.
- The enumerator is advanced to the next element by evaluating the function invocation `e.MoveNext()`.
- If the value returned by `e.MoveNext()` is `true`, the following steps are performed:
  - The current enumerator value is obtained by evaluating the property access `e.Current`, and the value is converted to the type of the iteration variable by an explicit conversion (§??). The resulting value is stored in the iteration variable such that it can be accessed in the statement.
  - Control is transferred to the statement. When and if control reaches the end point of the statement (possibly from execution of a `continue` statement), another `for each` iteration is performed, starting with the step above that advances the enumerator.
- If the value returned by `e.MoveNext()` is `false`, control is transferred to the end point of the `for each` 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
```

An 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*.

Need to add text to indicate the circumstances under which the modreq IsUdtReturn shall be emitted (i.e., ref class type retruned by value). Point to that modreq's spec.[[#134]]

### 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*.

### 16.3.5 The throw statement

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

## 16.4 The try statement

A program that attempts to `throw nullptr` is ill-formed.

In the grammar specified by Standard C++ (§15), the *try-block* and *function-try-block* productions have been extended 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*]

A program is ill-formed if it:

- uses a `break`, `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*.

# 17. Namespaces

To be added. [[#47]]

# 18. 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 (§19), ref classes (§20), value classes (§21), and interfaces (§24), are specified in the clauses that define those types. [*Note:* A summary of that support is shown in the following table:

This table and corresponding sections should include Special Member Functions (SMFs) like destructors, copy constructors, default constructors, assignment operators, conversion to special bool, handle equality. Many of these are not supported for value classes.[[#135]]

| Feature | Native class | Ref class | Value class | Interface |
|---|---|---|---|---|
| Class modifier | X | X | X | |
| Reserved member names | X | X | X | X |
| Function modifiers | X | X | X | n/a |
| Override specifier | X | X | X | n/a |
| Parameter arrays | X | X | X | X |
| Properties | | X | X | X |
| Events | | X | X | X |
| Static operators | X | X | X | X |
| Static constructor | | X | X | X |
| Literal field | | X | X | X |
| Initonly field | | X | X | X |
| Delegate definitions | X | X | X | X |
| Member of delegate type | | X | X | |

*end note*]

## 18.1 Class definitions

In the C++ Standard (§9), a *class-specifier* is used to define a class. This grammar has been extended to accommodate the addition of public and private classes, as follows:

*class-specifier:*
*top-level-type-visibility*$_{opt}$ *class-head* **{** *member-specification*$_{opt}$ **}**

*top-level-type-visibility* is described in §12.4

To accommodate the addition of initonly and literal fields, delegates, events, and properties, the syntactic class *member-declaration* in the C++ Standard (§9.2) has been extended, as follows:

*member-declaration:*
*attributes*$_{opt}$ *initonly-or-literal*$_{opt}$ *decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$ **;**
…
*delegate-definition*
*event-definition*
*property-definition*

*initonly-or-literal:*
    `initonly`
    `literal`

Attributes are described in §28, initonly fields are described in §18.10, literal fields in §18.9, delegates in §26, events in §18.5, and properties in §18.4.

### 18.1.1 Class modifiers

To accommodate the addition of sealed and abstract classes, the grammar for *class-head* in the C++ Standard (§9) has been extended to include an optional sequence of class modifiers, as follows:

*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-modifiers:*
    *class-modifier*
    *class-modifiers class-modifier*

*class-modifier:*
    `abstract`
    `sealed`

If the same modifier appears multiple times in a class definition, 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*]

The `abstract` and `sealed` modifiers are discussed in §18.1.1.1 and §18.1.1.2, respectively.

#### 18.1.1.1 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*]

#### 18.1.1.2 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*]

## 18.2 Reserved member names

To facilitate the underlying C++/CLI runtime implementation, for each member definition that is a property or event, the implementation must reserve several names based on the kind of the member definition (§18.2.1, §18.2.2). A program is ill-formed if it contains a class that declares a member whose name matches any of these reserved names, even if the underlying runtime implementation does not make use of these reservations. If a particular name is reserved within a class, that name is also reserved in all classes that derive from that class.

The reserved names do not introduce definitions, thus they do not participate in member lookup.

[*Note:* The `new` modifier cannot be used to circumvent the restriction that a member with a reserved name shall not be declared. *end note*]

[*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 for functions (§18.2.3).

### 18.2.1 Member names reserved for properties

For a scalar or named indexed property `P` (§18.4), 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 default indexed property (§18.4), 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.

Need to address the following: C++/CLI uses the System::Reflection::DefaultMemberAttribute attribute to specify that something other than the default name, “Item”, should be used. Given that, the text describes what happens if no name is chosen; that is, Item is used by default. Once the name has been set with DefaultMember, it cannot be changed in a derived class. If two interfaces have different DefaultMember attributes, implementing both interfaces is ill-formed.[[#136]]

### 18.2.2 Member names reserved for events

For an event `E` (§18.5), the following names are reserved:

```
add_E
remove_E
raise_E
```

### 18.2.3 Member names reserved for functions

For CLI types, the following name is reserved:

```
Finalize
```

## 18.3 Functions

Extend the grammar to accommodate attributes on functions.[[#137]]

The addition of overriding specifiers and function modifiers requires a change to the Standard C++ grammar for *direct-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*]

One of the productions for the Standard C++ grammar for *member-declarator* (§9.2) has been extended, as follows:

override-specifier should support 0 for compatibility with pure-specifier.[[Ed.]]

*member-declarator:*
*declarator* *function-modifiers*$_{opt}$ *override-specifier*$_{opt}$

*function-modifiers:*
*function-modifier*
*function-modifiers* *function-modifier*

*function-modifier:*
`abstract`
`new`
`override`
`sealed`

*function-modifier*s are discussed in the following subclauses: `abstract` in §18.3.3, `new` in §18.3.4, `override` in §18.3.1, and `sealed` in §18.3.2. *override-specifier* is discussed in §18.3.1.

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*.

The Standard C++ grammar for *parameter-declaration-clause* (§8.3.5) has been extended to include support for passing parameter arrays, as follows:

*parameter-declaration-clause:*
…
*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 §18.3.6.

### 18.3.1 Override functions

The Standard C++ grammar for *direct-declarator* has been extended (see §18.2.3) to allow the function modifier `override` as well as override specifiers.

*override-specifier:*
`=` *overridden-name-list*

*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 is ill-formed. [*Note:* A programmer can eliminate the diagnostic by using explicit or named overriding. *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. (Each named override corresponds exactly to a single MethodImpls in metadata. See "Explicit method overrides" in CLI Partition II.)

If either the *function-modifier* `override` or an *override-specifier*, or both, are present in the derived class function declaration, no implicit overriding takes place. [*Example:*

```
struct A {
   virtual void f() abstract;
};

struct B {
   virtual void f() abstract;
};

struct D : A, B {
   virtual void f();                    // overrides A::f and B::f
};

struct E : A, B {
   virtual void g() = B::f;             // overrides B::f only, E is
abstract
};

struct F : A, B {
   virtual void f() override;           // overrides A::f and B::f
};
```

*end example*]

Explain the difference between using 'override' and '= function-name'; one creates an .override directive in CIL, the other does not. [[#48]]

[*Note:* A member function declaration containing the *function-modifier* `override` or an *override-specifier* shall explicitly be declared `virtual` (§18.2.3). *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 and shall include that function's base class name. 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 covariant.

[*Example:*

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

struct B {
   virtual void f();
};

struct D : A, B {
   virtual void g() = A::f, B::f;   // override A::f and B::f
};
```

*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. [*Example:*

```
struct A {
   virtual void f();
   virtual void g();
   virtual void x();
};

struct B {
   virtual void f();
   virtual void g();
};

struct D : A, B {
   virtual void x() override = A::f;   // x overrides A:;f
   virtual void y() = A::g, B::f;      // y overrides A::g and B::f
};
```

*end example*]

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:*

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

struct B {
   virtual void f();
   virtual void g();
};

struct D : A, B {
   virtual void g() = B::f;
   virtual void f();          // error, would override A::f and B::f, but
                              // B::f is already overridden
   virtual void f() override = B::g;
                              // error, B::g is overridden twice,
                              // once by the explicit override, and
                              // once by the named override.
   virtual void f() = B::f;   // error, B::f is overridden twice,
                              // once by the implicit override, and
                              // once by the named override.
};
```

*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:*

```
struct D : B1, B2 {
   void f() = B1::f { /*…*/ }    // ok
   void f() = B2::f { /*…*/ }    // error, duplicate declaration
};
```

*end example*]

A function can both hide and override at the same time: [*Example:*

```
struct A {
   virtual void f();
};
struct B {
   virtual void f();
};
struct D : A, B {
   virtual void f() new = A::f;
};
```

The presence of the `new` function modifier (§18.3.4) indicates that `D::f` does not override any method `f` from its bases classes. The named override then goes on to say that `D::f` actually overrides just one function, `A::f`. *end example*]

A member function that is an explicit override cannot be called directly (except with explicit qualification) or have its address taken. [*Example:*

```
struct I {
   virtual void v();
};
struct J {
   virtual void w();
};
struct A : I, J {
   virtual void f() = I::v, J::w;
};
struct C : A {
  virtual void g() = I::v;
  virtual void h() = J::w;
};
void Test(A* pa) {// pa could point to an A, a C, or something else
   pa->f();       // ambiguous: I::v or J::w?
   pa->v();       // ok, virtual call
   pa->w();       // ok, virtual call
   pa->I::v();    // ok if I::v is implemented, nonvirtual call to I::v
   pa->J::w();    // ok if J::w is implemented, nonvirtual call to J::w
   pa->A::v();    // ok if I::v is implemented, nonvirtual call to I::v
   pa->A::w();    // ok if J::w is implemented, nonvirtual call to J::w
   pa->A::f();    // ok (classes derived from A might need to do this,
                  //  and there's no ambiguity in this case)
}
```

*end example*][*Rationale:* Even though technically it is possible to allow a call to such an `f` when the type of the Object is statically known to be an `A`, for example in:

```
A a;
a.f();               // ambiguous (even though it could work)
```

there does not seem to be sufficient utility to offset the user confusion about "When can I do this and when can't I?" *end rationale*]

If a destructor or finalizer (§??) contains an override specifier, the program is ill-formed.

### 18.3.2 Sealed function modifier

A virtual member function marked with the *function-modifier* `sealed` cannot be overridden in a derived class. [*Example:*

```
struct B {
   virtual int f() sealed;
};

struct D : B {
   virtual int f();     // error: cannot override a sealed function
};
```

*end example*]

[*Note:* A member function declaration containing the *function-modifier* `sealed` shall explicitly be declared `virtual` (§18). *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 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*]

### 18.3.3 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 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` (§18). *end note*]

### 18.3.4 New function modifier

A member function declaration containing the *function-modifier* `new` shall not contain an *override-specifier*.

The `new` function modifier corresponds exactly to the CLI's predefined attribute ***newslot*** (see the CLI Standard, Partition II, an excerpt of which is shown as a note below.). A function's metadata will have the newslot attribute if that function's declaration included the `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. It can, however, 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() { System::Console::WriteLine("B::F"); }
   virtual void G() { System::Console::WriteLine("B::G"); }
};

ref struct D : B {
   virtual void F() new { System::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:

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

struct B {
   virtual void f();
};

struct D : A, B {
   virtual void f() new = A::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 (§18.3.1) then goes on to say that `D::f` actually overrides just one function, `A::f`. The net result is that `A::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*]

[*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*]

### 18.3.5 Function overloading

The C++ Standard (§13.3.2) has been extended to incorporate parameter arrays (§18.3.6), as follows:

> "For every parameter array function, two signatures are submitted to the overload candidate set: the expanded form and the exact signature."

### 18.3.6 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*

Re the following: For functions outside CLI types, if they happen to have a parameter array, it is okay to have a default parameter. That parameter can be any Array -- the parameter array part of it is just ignored and instead for the purposes of the default parameter is just a plain Array.

A *parameter-array* consists of an optional set of *attributes* (§28), an ellipsis punctuator, and a *parameter-declaration*. A parameter array declares a single parameter of the given Array type with the given name. The Array type of a parameter array must be a single-dimensional Array type (§23.1). In a function invocation, either a parameter array permits a single argument of the given Array type to be specified, or it permits zero or more arguments of the Array element type to be specified. The program is ill-formed if the *parameter-declaration* contains an *assignment-expression*.

```
void f(... array<Object^>^);

int main() {
   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, '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 in its expanded form, the invocation is processed exactly as if an Array creation expression with an Array initializer (§??) was inserted around the expanded parameters. [*Example:* Given the declaration

```
void F(int x, int y, ... array<Object^>^ args);
```

the following invocations of the expanded form 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});
```

In particular, `nullptr` is passed when there are zero arguments given for the parameter array. *end example*]

Parameter array parameters can be passed to functions that take non-parameter 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<type>` can be passed to a function having a parameter `... array<type>`. In the case of passing an `array<Object^>` argument `A` to a parameter `P` (declared using `... array<Object^>`), `P` binds to `A` (that is, `P` is not an Array whose first `Object^` element refers to `A`).

Parameter arrays can contain either native or CLI type elements. [*Example:*

```
void g(... array<Object^>% v);   // CLI type held by ^
g(1, 2, "abc");                        // creates a container of 3 boxed
                                       // Objects, having type Int32,
                                       // Int32, and String.
void h(... array<std::string>% a);     // native type held by value
h("abc", "def", "xyzzy", string2);     // creates a container of 4
strings
```

*end example*]

## 18.4 Properties

1. Can a trivial (scalar) property be static or virtual? Yes
2. Does a property member always make a class a non-POD? No
3. Can the value of a property be passed by reference or by const reference even if the type of the property is not a reference? No
4. Is compound assignment to the result of a property access allowed? Yes, a += b allowed, but a = b = c is not because CLS require that the setter have a void return type.
5. Can accessor functions be cv-qualified (examples in this paper const-qualify getters)? No
6. Can a property have reference type? No for CLS properties; otherwise, Yes.

[[Ed.]]

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 scalar field-like access to an Object or class. 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 an 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 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:

Extend declarator-id's by adding a new production that allows default. [[#50]]

*property-definition:*
*attributes*$_{opt}$ *property-modifiers* *simple-type-specifier* *declarator*
*property-indexes*$_{opt}$ *function-modifiers*$_{opt}$ *override-specifier*$_{opt}$
`{` *accessor-specification* `}`
*attributes*$_{opt}$ *property-modifiers* *simple-type-specifier* *declarator*
*function-modifiers*$_{opt}$ *override-specifier*$_{opt}$ `;`

*property-modifiers:*
*property-modifier*
*property-modifiers* *property-modifier*

*property-modifier:*
`property`
`static`
`virtual`

*property-indexes:*
`[` *indexer-parameter-list* `]`

*indexer-parameter-list:*
*indexer-parameter-declaration*
*indexer-parameter-list* `,` *indexer-parameter-declaration*

*indexer-parameter-declaration:*
*type-specifier*

The grammar for indexer-parameter-declaration does not allow handles or pointers, but full declarators are not needed. The grammar should allow a simpler sequence of ptr-operator. [[#51]]

A *property-definition* can include a set of *attributes* (§28), *property-modifier*s (§18.4.1, §18.4.3), *property-indexes*, *function-modifiers* (§18.2.3), and an *override-specifier* (§18.3.1). It must include the *property-modifier* `property`.

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* ending with a semicolon (as opposed to brace-delimited *accessor-specification*) defines a ***trivial scalar property*** (§18.4.4). [*Note:* There is no such thing as a trivial indexed property. *end note*]Need to write up the restrictions on trivial properties.[[#138]]

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 is not permitted on a default indexed property definition. (Default indexed properties are introduced later in this subclause.)

The *simple-type-specifier* of a scalar property definition specifies the type of the scalar property introduced by the definition, and the *identifier* specifies the name of the scalar property. The *simple-type-specifier* of an indexed property definition specifies the element type of the indexed property introduced by the definition.

*property-name* specifies the name of the property. For an indexed property, if *property-name* is `default`, that property is a ***default indexed property***. If *property-name* is *identifier*, that property is a ***named indexed property***.

We probably should say something about the reserved names get_Item and set_Item, and their relationship with default indexed properties. Also, add a forward pointer to the corresponding attribute.[[#139]

The *accessor-specification* declares the accessor functions (§18.4.2) 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`.

The address of an accessor function can be taken and yields a pointer-to-member of the enclosing type. However, it is not possible to bind a pointer-to-member value to a property. [*Note:* A property is a group of one or more accessor functions, not an Object. *end note*]

An indexed property cannot have the same name as a scalar property. Overloading of indexed properties on different index parameters is allowed, as long as none has the same name as a scalar property.

### 18.4.1 Static and instance properties

When a property definition includes a `static` modifier, the property is said to be a ***static property***. [*Note:* An 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 writing `static` on such a function is allowed but redundant. All accessor functions in an instance property are instance accessor functions. [*Example:*

```
struct C {
   static property C* MyStaticProperty { /* … */ } // static property
   property int default[int k] { /* … */ };        // instance property
};
```

*end example*]

[*Note:* Like a field, when a static property is referenced using the form `E::M`, `E` must denote a type that has a property `M`. When an instance property is referenced using the form `E.M`, `E` must 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*]

### 18.4.2 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:*
*decl-specifier-seq*$_{opt}$ *member-declarator-list*$_{opt}$ `;`
*function-definition* `;`

A property must have at least one accessor function. The name of a property accessor function must be either `get` or `set`. 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.

If a property has the `static` modifier, all of its accessor functions are implicitly `static`; nevertheless, declaring `static` on one or more of those accessor functions is allowed but redundant.

If a property is abstract, the accessor functions of the property can be abstract. If an accessor function is not declared abstract, it must be defined. If any accessor function of a property is declared abstract, the property must also be declared abstract.

The get accessor function of a scalar property takes no parameters and its return type shall match exactly the type of the property, simple-type-specifier. A get accessor function shall not return an array. For an indexed property, the parameters of the get accessor function shall correspond exactly to the types of the property's *property-indexe*.

This subclause only covers how the accessor functions must be defined. The expressions clause needs to cover the rewrite rules that call these functions. [[#52]]

The set accessor function of a scalar property has one parameter that corresponds exactly to the type of the property, simple-type-specifier. 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 the last parameter, which shall correspond exactly to the type of the property, simple-type-specifier. 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 specifies a narrower access than the property's accessibility for that accessor function. *access-specifier*s on accessor functions specify access for those accessor functions only; they have 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.

[*Note:* If the get and set accessor functions in a read-write property have different implicit or explicit *access-specifier*s, that property is not CLS-compliant. *end note*]

[*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 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*]

In the paragraph above, add a cross-reference to the rewrite rules for properties and events. (They will be somewhere in the expressions clause.) [[Ed]]

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. [*Example:* In the example

```
struct A {
   property int P {
      void set(int value) {…}
   }
};

struct B : A {
   property int P {
      int get() {…}
   }
};
```

the `P` property in `B` hides the `P` property in `A` with respect to both reading and writing. Thus, in the statements

```
B b;
b.P = 1;                    // Error, B.P is read-only
b.A::P = 1;                 // Ok, reference to A.P
```

the assignment to `b.P` causes the program to be ill-formed, since the read-only `P` property in `B` hides the write-only `P` property in A. Note, however, that a cast can be used to access the hidden `P` property. *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*]

Add some discussion of how accesses to properties are rewritten into accessor functions. This should be covered in rewrite rules in the expressions clause. Note that access checking for whether a property can be written to or read to is done after rewriting and overload resolutions. [[#116]]

Accessor functions can be defined inline or out-of-class. [*Example:*

```
public class point {
private:
   int Xor;
   int Yor;

public:
   property int X {
      int get() { return Xor; }                    // inline definition
      void set(int value);                         // declaration only
   }

   property int Y {
      int get();                                   // declaration only
      void set(int value) { return Yor = value; }  // inline definition
   }
   …
};

void point::X::set(int value) { Yor = value; }
int point::Y::get() { return Yor; }
```

*end example*]

The qualified name of a property needs to be described somewhere. Once that happens, how an out-of-class definition is done will already be covered by existing rules. [[#117]]

### 18.4.3 Virtual, sealed, abstract, and override accessor functions

A `virtual` property definition specifies that the accessor functions of the property are virtual. Declaring `virtual` on an accessor function of a virtual property is allowed but redundant. If the `virtual` modifier appears on every accessor function in a property not itself having such a modifier, then that modifier applies implicitly to the property.

A `sealed` property definition specifies that the accessor functions of the property are sealed. A property definition containing the *function-modifier* `sealed` shall explicitly be declared `virtual`. Use of this modifier prevents a derived class from further overriding the property. Declaring `sealed` on an accessor function of a sealed property is allowed but redundant. If the `sealed` modifier appears on every accessor function in a property not itself having such a modifier, then that modifier applies implicitly to the property.

An `abstract` property definition specifies that the accessor functions of the property are abstract and virtual, but does not provide an actual implementation of the accessor functions. Instead, non-abstract derived classes are required to provide their own implementation for the accessor functions by overriding the property. A property definition containing the *function-modifier* `abstract` shall explicitly be declared `virtual`. All of the accessor functions of an abstract property can also individually contain an `abstract` and/or `virtual` modifier; however, such modifiers are redundant. If the `abstract` modifier appears on every accessor function in a property not itself having such a modifier, then that modifier applies implicitly

to the property. A virtual property can have abstract accessor functions, and the property need not be explicitly declared abstract.

[*Example:*

```
struct B {
   virtual property string Name {      // virtual property
      virtual string get() abstract;   // property is implicitly abstract
   }
};

struct D : B {
   virtual property string Name sealed { /*…*/ }   // Name is now 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 abstract {
   property int Size { /*…*/ };  // (implicit) abstract property
   virtual property string Name abstract = 0 { /*…*/ };
      // "virtual", abstract" and "= 0"
      // 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. The accessor functions of such a property are also abstract.

[*Note:* Abstract property definitions are only permitted in abstract classes (§18.1.1.1). *end note*]

The accessor functions of an inherited virtual property can be overridden in a derived class by including a property definition that specifies an `override` modifier or an *override-specifier* (§18.3.1). This is known as an ***overriding property definition***. An overriding property definition does not declare a new property. Instead, it simply specializes the implementations of the accessor functions of an existing virtual property.

[*Example:*

```
struct B1 {
   virtual property string Name { /*…*/ }
};

struct B2 {
   virtual property string MyName { /*…*/ }
};

struct D : B1, B2 {
   // override both
   virtual property string HelloIAm = B1::Name, B2::MyName { /*…*/ }
};
```

*end example*]

An accessor function can override accessor functions in other properties; it can also override non-accessor functions. [*Example:*

```
struct B {
   virtual property string Name {
      string get();
      void set(string value);
   }
};

struct C {
   virtual string getLabel();
};
```

```
struct D : B, C {
   virtual property string MyName = B::Name {
      string get() = C::getLabel;   // implicitly overrides Name::get and
   }                                // explicitly overrides C::getLabel
};
```

*end example*]

An overriding property definition must specify wider accessibility modifiers 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 must 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 must 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 §18.3.2, §18.3.1, and §18.3.3 of this Standard apply as if accessor functions were functions of a corresponding form:

[*Example:* In the example

```
class A abstract {
   int y;

public:
   virtual property int X {
      int get() { return 0; }
   }

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

   virtual property int Z abstract {
      int get();
      void set(int value);
   }
};
```

`X` is a virtual read-only property, `Y` is a virtual read-write property, and `Z` is an abstract read-write property.

### 18.4.4 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:*

```
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 is automatically allocated with the name of that storage being unspecified, but in the implementer's namespace. [*Example:* A compiler might treat the above trivial scalar property definition as if it was written like the following:

```
struct S {
   property int P {
      int get() { return __P; }
      void set(int value) { __P = value; }
   }
private:
   int __P;
};
```

*end example*]

## 18.5 Events

An ***event*** is a member that enables an Object or class to provide notifications. Clients can add a delegate to an event, so that the Object will invoke that delegate. Events are declared using *event-definition*s:

*event-definition:*
    *attributes*$_{opt}$ *event-modifiers* *event-type* *identifier*
        *function-modifiers*$_{opt}$ *override-specifier*$_{opt}$ `{` *accessor-specification* `}`
    *attributes*$_{opt}$ *event-modifiers* *event-type* *identifier*
        *function-modifiers*$_{opt}$ *override-specifier*$_{opt}$ `;`

*event-modifiers:*
    *event-modifier*
    *event-modifiers* *event-modifier*

*event-modifier:*
    `event`
    `static`
    `virtual`

An *event-definition* can include a set of *attributes* (§28), *property-modifier*s (§18.4.1, §18.4.3), *function-modifiers* (§18.2.3, §18.4.3), and an *override-specifier* (§18.3.1). It must include the *event-modifier* `event`.

The *event-type* of an event definition shall be a delegate type, and that type shall be at least as accessible as the event itself. *identifier* designates the name of the event.

The production event-type has not yet been defined. The syntactic category of this element needs to be reviewed.[[#140]]

The *accessor-specification* declares the accessor functions (§18.5.2) 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.

An *event-definition* ending with a semicolon (as opposed to a brace-delimited *accessor-specification*) defines a ***trivial event*** (§18.5.4). 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 Button : Control {
   event EventHandler^ Click;
};

public ref class LoginDialog : Form
{
   Button^ OkButton;
   Button^ CancelButton;

public:
   LoginDialog() {
      OkButton = gcnew Button(…);
      OkButton->Click += gcnew EventHandler(&OkButtonClick);
      CancelButton = gcnew Button(…);
      CancelButton->Click += gcnew EventHandler(&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 `Button` instances and attaches event handlers to the `Click` events. *end example*]

The address of an event accessor function can be taken and bound to a suitably typed pointer-to-member function (subject to the usual C++ rules, such as that the calling code must have access to the function's name). However, it is not possible to bind a pointer-to-member Object to an event. [*Note:* An event is a group of one or more accessor functions, not an Object. *end note*]

### 18.5.1 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***.

### 18.5.2 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 three *function-definition*s:

It is a bit strange to define grammar productions for these functions. We probably should either make these terms (and change the style accordingly) or just call them the add function, remove function, and raise function.[[#141]]

- one for a function called `add`, herein called the *add-accessor-function*,
- one for a function called `raise`, herein called the *raise-accessor-function*, and
- one for a function called `remove`, herein called 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 one of *add-accessor-function* and *remove-accessor-function*.

*add-accessor-function* and *remove-accessor-function* shall each take one parameter, of type *event-type*, and their return type shall be `void`.

The parameter list of *raise-accessor-function* shall correspond exactly to the parameter list of *event-type*, and its return type shall be the return type of *event-type*.

[*Note:* Trivial envents 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 function is called.

[*Example:* … *end example*] [[Ed]]

### 18.5.3 Virtual, sealed, abstract, and override accessor functions

A `virtual` event declaration specifies that the accessor functions of that event are virtual. The `virtual` modifier applies to all accessor functions of an event.

An `abstract` event declaration specifies that the accessor functions of the event are virtual, but does not provide an actual implementation of the accessor functions. Instead, non-abstract derived classes are required to provide their own implementation for the accessor functions by overriding the event.

An event declaration that includes both the `abstract` and `override` modifiers specifies that the event is abstract and overrides a base event. The accessor functions of such an event are also abstract.

[*Note:* Having an abstract event makes the enclosing class abstract. *end note*] 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.

An overriding event declaration can include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the event. The accessor functions of a sealed event are also sealed.

An event with the new modifier introduces a new event that does not override an event from a base class. Make sure the complete specification is provided in the clause for the new modifier.[[#142]] Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessor functions behave exactly like virtual, sealed, override and abstract functions.

When a trivial event overrides an event, the trivial event's `raise` is implicitly declared and defined.

### 18.5.4 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*]

Within the class that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must be trivial. Such an event can be used in any context that permits a field. The field contains a delegate, which refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains `nullptr`.

[*Example:* In the example

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

public ref class Button : Control {
public:
   event EventHandler^ Click;
   void Reset() {
      Click = nullptr;
   }

protected:
   void OnClick(EventArgs^ e) {
      Click(this, e);  // raise tests for nullptr
   }
};
```

`Click` is used as a field within the `Button` class. As the example demonstrates, the field can be examined, modified. The `OnClick` function in the `Button` class "raises" the `Click` event.

Outside the declaration of the `Button` class, the `Click` member can only be used on the left-hand side of the `+=` and `-=` operators, as in

```
b->Click += gcnew EventHandler(…);
```

which appends a delegate to the invocation list of the `Click` event, and

```
b->Click -= gcnew EventHandler(…);
```

which removes a delegate from the invocation list of the `Click` event. *end example*]

When compiling a trivial event, the compiler automatically creates storage to hold the delegate, and creates accessor functions for the event that add event handlers to, and remove them from, the delegate field. The compiler also automatically generates a raise accessor function. The access-specifier for the generated `add` and remove accessor functions is the same as that for the whole event. The access-specifier for the generated raise accessor function is `protected`. In order to be thread-safe, the addition and removal operations shall be done while holding the lock on the containing Object for an instance event, or the type Object for a static event. Such a lock is specified using the attribute

`MethodImpl(MethodImplOptions::Synchronized)`. The compiler-generated raise accessor function shall not have this attribute.

[*Note:* Thus, an instance event declaration of the form:

```
delegate int D(int);
ref class X {
public:
   event D^ Ev;
};
```

could be compiled to something equivalent to:

```
ref class X {
   D^ __Ev;       // field to hold the delegate
public:
   event D^ Ev {
      [MethodImpl(MethodImplOptions::Synchronized)]
      void add(D^ value) {
         __Ev += value;
      }
      [MethodImpl(MethodImplOptions::Synchronized)]
      void remove(D^ value) {
         __Ev -= value;
      }
protected:
      int raise(int arg) { return __Ev(arg); }
   }
};
```

Within the class `X`, references to `Ev` are compiled to reference the hidden field `__Ev` instead. (The name "`__Ev`" is arbitrary; the hidden field could have any name or no name at all.)

Similarly, a static event declaration of the form:

```
delegate int D(int);
ref class X {
public:
   static event D^ Ev;
};
```

could be compiled to something equivalent to:

```
ref class X {
   static D^ __Ev;      // field to hold the delegate
public:
   static event D^ Ev {
      [MethodImpl(MethodImplOptions::Synchronized)]
      void add(D^ value) {
         __Ev += value;
      }
      [MethodImpl(MethodImplOptions::Synchronized)]
      void remove(D^ value) {
         __Ev -= value;
      }
   protected:
      int raise(int arg) { return __Ev(arg); }
   }
};
```

*end note*]

### 18.5.5 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.

Events without a raise accessor function cannot be invoked using function call syntax. Instead, the delegate's `Invoke` function must be called directly.

## 18.6 Static operators

Add examples throughout this clause. [[Ed]]

To support the definition of operators in CLI types, 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 relaxed to allow static member functions:

> (The restriction below does not apply to non-static member operators – that need not have a parameter of the type of the class.)[[#143]]"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."

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 or static 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 or static 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: `operator=` (§13.5.3), `operator()` (§13.5.4), `operator[]` (§13.5.5), and `operator->` (§13.5.6). *end note*]

### 18.6.1 Homogenizing the candidate overload set

Provide an example.[[#144]]

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 function (including static member functions) 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 exactly to the one member function from which the signatures were created.

[*Rationale:* This allows functions to be called using variables that have the raw type 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 namespace scope.) *end rationale*]

### 18.6.2 Operators on Handles

Unlike pointers, some user-defined operators can be defined for 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 changed, as follows:

> "Thus, for any prefix unary `operator@`, `@x` can be interpreted as ~~either~~ `x->operator@()` if `x` is a handle, `x.operator@()` if `x` is not a handle, or `operator@(x)`."

Standard C++ (§13.5.2/1) is changed, as follows:

> "Thus for any binary `operator@`, `x@y` can be interpreted as ~~either~~ `x->operator@(y)` if `x` is a handle, `x.operator@(y)` if `x` is not a handle, or `operator@(x,y)`."

[*Note:* In C++/CLI, equality operators for handles behave as if they were compiler-generated or user-defined operators. *See* §18.6.6.1. *end note*]

The rules in Standard C++ (§13.5.3/1) continue to apply—an assignment operator shall be a 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 changed, 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 changed, 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 does not apply to a handle. Like a pointer, `x->y` is defined as `(*x).y`. A member access to a handle never invokes the user defined member access operator.

[*Note:* The increment and decrement operators described in Standard C++ (§13.5.7), have significant differences from the CLS increment and decrement operators. (*See* §18.6.3 for details.) *end note*]

### 18.6.3 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 argument.
  - The saved value of `x` is the result of the expression.
- Otherwise:
  - The operator is processed as specified by Standard C++.

Add an example.[[Ed.]]

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 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.
  - The saved value from the operator invocation is the result of the expression.
- Otherwise:
  - The operator is processed as specified by Standard C++.

Add an example. [[Ed.]]

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.

Add an example. [[Ed.]]

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:
   // ...
   static IntVector^ operator++(IntVector^ iv) { /*...*/ }
};
```

```
int main() {
   IntVector^ iv1 = gcnew IntVector;
   IntVector^ iv2;

   iv2 = iv1++;
     // equivalent to:
     //   IntVector^ __temp = iv1;
     //   iv1 = IntVector::operator++( iv1 );
     //   iv2 = __temp;

   iv2 = ++iv1;
     // equivalent to:
     //   iv1 = IntVector::operator++( iv1 );
     //   iv2 = iv1;
}
```

Note: 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*]

### 18.6.4 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`, and the transformed expression is interpreted with the rules specified so far. Identify when synthesis would and would not occur. [[#56]]

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.

### 18.6.5 Naming conventions

During compilation, the name of every operator function is the C++ identifier used in source code for that function. For example, the addition operator's identifier is `operator+`. When the compiler emits the program to an assembly, the metadata name for the operator function is the CLS-compliant name as specified herein.

The CLS-compliant name for the operator function is only used in the compiled assembly. A program shall not refer to the CLS-compliant name given to the operator function. When the compiler imports functions from metadata, it shall rewrite the CLS-compliant name into the respective C++ operator function identifier. Likewise, when the compiler emits metadata for the program, it translates the C++ operator function identifier to the respective CLS-compliant name.

A C++ program shall not declare nor define a function using one of the CLS-compliant identifiers referred to herein.

The CLS recommends certain operators upon which CLS consumer and producer languages can agree. The set of CLS-compliant operators overlaps with the set of operators supported by C++ (see Partition I, §9.3, of the CLI Standard) as described in §18.6.5.1. The C++ operators that do not overlap with the CLS-compliant operators are known as C++-dependent operators (§0).

All operator functions, regardless of whether they are CLS-compliant operators or C++-dependent operators, shall be marked as SPECIALNAME functions in the metadata.

#### 18.6.5.1 CLS-compliant operators

An operator is CLS-compliant when the following conditions occur:

1. The operator function is one listed in either Table 18-1: CLS-Recommended Unary Operators or Table 18-2: CLS-Recommended 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 by handle. The handle shall not be passed by reference.

If the above criteria are not met, the operator function is C++-dependent (§18.6.5.4). Table 18-1: CLS-Recommended Unary Operators and Table 18-2: CLS-Recommended Binary Operators list the name that shall be given to the function used to represent the operator function in an assembly.

When importing a class from an assembly, each static member function with a name listed in Table 18-1: CLS-Recommended Unary Operators and Table 18-2: CLS-Recommended Binary Operators shall be renamed with its corresponding C++ identifier for the operator function.

**Table 18-1: CLS-Recommended Unary Operators**

| Function Name in Assembly | C++ Operator Function Name |
|---|---|
| op_UnaryNegation | operator- |
| op_UnaryPlus | operator+ |
| op_LogicalNot | operator! |
| op_AddressOf | operator& |
| op_OnesComplement | operator~ |
| op_PointerDereference | operator* |

**Table 18-2: CLS-Recommended Binary Operators**

| Function Name in Assembly | C++ Operator Function Name |
|---|---|
| op_Decrement | operator-- |
| op_Increment | operator++ |
| op_Addition | operator+ |
| op_Subtraction | operator- |
| op_Multiply | operator* |
| op_Division | operator/ |
| op_Modulus | operator% |
| op_ExclusiveOr | operator^ |
| op_BitwiseAnd | operator& |
| op_BitwiseOr | operator\| |
| op_LogicalAnd | operator&& |
| op_LogicalOr | operator\|\| |
| op_LeftShift | operator<< |
| op_RightShift | operator>> |
| op_Equality | operator== |
| op_GreaterThan | operator> |
| op_LessThan | operator< |
| op_Inequality | operator!= |
| op_GreaterThanOrEqual | operator>= |
| op_LessThanOrEqual | operator<= |
| op_Comma | operator, |

### 18.6.5.2 Non-C++ operators

The CLS recommends some 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 §E.1. *end note*]

The ability to define operator true and operator false will be provided. [[#57]]

| Function Name in Assembly | C++ Operator Function Name |
|---|---|
| `op_True` | Not yet defined[[#145]] |
| `op_False` | Not yet defined |

### 18.6.5.3 Assignment operators

Given that 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, the C++ compiler does not generate or consume CLS assignment operators (as listed in Table 18-3: CLS-Recommended Assignment Operators). As such, user-defined functions with names from Table 18-3: CLS-Recommended Assignment Operators are not given special treatment.

**Table 18-3: CLS-Recommended Assignment Operators**

| Function Name in Assembly | 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 |

### 18.6.5.4 C++-dependent operators

If an operator function does not match the criteria for a CLS-compliant operator, as listed in §18.6.5.1, the operator is C++-dependent. Table 18-4: C++-Dependent Unary Operators and Table 18-5: C++-Dependent Binary Operators list the metadata name for each function.

When importing functions from an assembly, functions with the names listed in Table 18-4: C++-Dependent Unary Operators and Table 18-5: C++-Dependent Binary Operators shall be treated during compilation 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 is callable by the name it has in the assembly.

These operator names are, in most cases, those recommended by the CLS even though they are not CLS-compliant.

Some operator names listed below are not part of the CLS recommendations. These are `op_FunctionCall` and `op_Subscript`.

[*Note:* The postfix increment and decrement operators are identified in C++ via a dormant `int` parameter. Static member increment and decrement operators shall not have such a dormant `int` parameter. Instead, a single static increment and decrement operator is used for both pre and post operations. (*See* §18.6.3 for more details.) *end note*]

**Table 18-4: C++-Dependent Unary Operators**

| Function Name in Assembly | C++ Operator Function Name |
|---|---|
| op_UnaryNegation | operator- |
| op_UnaryPlus | operator+ |
| op_LogicalNot | operator! |
| op_AddressOf | operator& |
| op_OnesComplement | operator~ |
| op_PointerDereference | operator* |

**Table 18-5: C++-Dependent Binary Operators**

| Function Name in Assembly | C++ Operator Function Name |
|---|---|
| op_Addition | operator+ |
| op_Subtraction | operator- |
| op_Multiply | operator* |
| op_Division | operator/ |
| op_Modulus | operator% |
| op_ExclusiveOr | operator^ |
| op_BitwiseAnd | operator& |
| op_BitwiseOr | operator\| |
| op_LogicalAnd | operator&& |
| op_LogicalOr | operator\|\| |
| op_LeftShift | operator<< |
| op_RightShift | operator>> |
| op_Equality | operator== |
| op_GreaterThan | operator> |
| op_LessThan | operator< |
| op_Inequality | operator!= |
| op_GreaterThanOrEqual | operator>= |
| op_LessThanOrEqual | operator<= |
| op_MemberSelection | operator-> |
| op_PointerToMemberSelection | operator->* |
| op_Comma | operator, |
| op_Decrement | operator-- |
| op_Increment | operator++ |
| op_Assign | operator= |
| op_RightShiftAssignment | operator>>= |
| op_MultiplicationAssignment | operator*= |
| op_SubtractionAssignment | operator-= |
| op_ExclusiveOrAssignment | operator^= |
| op_LeftShiftAssignment | operator<<= |
| op_ModulusAssignment | operator%= |
| op_AdditionAssignment | operator+= |
| op_BitwiseAndAssignment | operator&= |
| op_BitwiseOrAssignment | operator\|= |
| op_DivisionAssignment | operator/= |
| op_FunctionCall | operator() |
| op_Subscript | operator[] |

### 18.6.6 Compiler-defined operators

#### 18.6.6.1 Equality

Reword this subclause similarly to the way special member functions are described. [[#58]]

Every type has an equality operator that works on handles. Every type behaves as if it had both a static `operator==` and `operator!=` where both arguments are handles to the containing type. That is, for type `T`, it is as if every type had the following operators:

```
static bool operator==(T^ lhs, T^ rhs);
static bool operator!=(T^ lhs, T^ rhs);
```

The purpose of these "as if" operators is to determine reference equality. Specifically, the return value of `operator==` is true if and only if both arguments are handles referring to the same Object. Conversely, the return value of `operator!=` is true if and only if both arguments are handles referring to different Objects.

If a type has a user-defined static `operator==` or `operator!=` with the same signature as the "as if" equality operators, then the user-defined operator is used. The user-defined operator is actually emitted to the assembly, whereas the "as if" operators are not.

Add another subclause to cover the compiler-generated conversion from handle to unspecified bool type. [[#59]]

## 18.7 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".

## 18.8 Static constructors

A ***static constructor*** is a function member that implements the actions required to initialize a ref or value class. A static constructor is declared just like an ordinary (that is, 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-list*.

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 (§10.5.3).

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 example

```
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 must be run before accessing any static members of `A`, and `B`'s static constructor must 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 X {
public:
   static X();       // static constructor declaration
   X();              // instance constructor declaration
   X(int) {…}        // inline instance constructor definition
};
static X::X() {…}    // out-of-class static constructor definition
X::X() {…}           // 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 can have any *access-specifier*. [*Note:* However, for security reasons, a static constructor should have a `private` *access-specifier*. *end note*]

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.

The static constructor cannot be explicitly invoked. A nontrivial static constructor is emitted as a `private` member of its class in metadata.

## 18.9 Literal fields

***Literal fields*** are defined by including the `literal` *storage-class-specifier*.

add literal to storage-class-specifier[[#146]]

Add grammar for literal-constant-initializer = Standard C++ constant-initializer + float/double + String + nullptr. [[#60]]

A ***literal field*** is a named compile-time constant rvalue having the type of the literal field and having the value of its literal-constant-initalizer.

Each *member-declarator* in the *member-declarator-list* shall contain a literal-*constant-initializer*. The *decl-specifier-seq* shall not contain a *cv-qualifier*.

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 one of the following types: a scalar type or `System::String`. A *literal-constant-expression* shall yield a value of the target type, or if the literal-constant-expression is not a string literal, it can be a value of a type that can be converted to the target type by a standard conversion sequence.

[*Note:* A *literal-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 *literal-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 (§18.10) can be used instead. [*Note:* The versioning semantics of `literal` and `initonly` differ (§18.10.2). *end-note*]

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 Color {red, white, blue};
   literal Color DefaultColor = 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[Size];
}
```

*end example*]

For a discussion of versioning and literal fields, see §18.10.2.

## 18.10 Initonly fields

***Initonly fields*** are defined by including the `initonly` storage-class-specifier.

add initonly to storage-class-specifier[[#147]]

Initialization of initonly fields shall occur only as part of their definition. Assignments (via an assignment operator or a postfix or prefix increment or decrement operator) to initonly fields shall occur only in an instance constructor or static constructor in the same class. [*Note:* Of course, such assignment could be done via a constructor's *ctor-initializer*. *end note*] (Although an initonly field can be assigned to multiple times in a given context, it shall be assigned in only one context.) Specifically, initialization of, and assignments to, initonly fields are permitted only in the following contexts:

- In the *constant-initializer* of a *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 its address or to bind it to a reference in any context, is ill-formed.

[*Example:*

```
ref class X {
   initonly static int svar1 = 1;// Ok
   initonly static int svar2;
   initonly static int svar3;

   initonly int mvar1 = 1;       // Error
   initonly int mvar2;
   initonly int mvar3;
public:
   static X(){
      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
   }
   X() : 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*]

### 18.10.1 Using static initonly fields for constants

A `static initonly` field is useful when a symbolic name for a constant value is desired.

Add a description that for any value class we have to make the copy before calling member functions. [[#62]]

### 18.10.2 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*]

## 18.11 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.

Any ref class can have a user-defined finalizer. The finalizer is run zero or more times by the garbage collector, as specified by the CLI.

Say more about finalizers (including Dispose/~T and Finalize/!T) and add some examples. [[#63]]

# 19. Native classes

The accessibility of a non-nested native class can optionally be specified via a *top-level-type-visibility* (§12.4).

A native class can optionally have a *class-modifiers* (§18.1.1).

## 19.1 Functions

A virtual member function in a native class can contain:

- the *function-modifier* `override`, or an *override-specifier*, or both (§18.3.1).
- the *function-modifier* `sealed` (§18.3.2).
- the *function-modifier* `abstract` (§18.3.3).

Member functions in a native class can optionally have a *parameter-array* (§18.3.6) in their *parameter-declaration-clause*.

## 19.2 Properties

Support for properties in native classes.

## 19.3 Static operators

Native classes support static operators (§18.6).

## 19.4 Instance constructors

## 19.5 Delegates

Native classes support *delegate-definition*s (§26); however, a native class shall not contain a field having a delegate type.

# 20. Ref classes

A ref class is a data structure known to the CLI runtime. It can contain fields, function members, and nested types.

## 20.1 Ref class declarations

A *reference-class-declaration* introduces a declaration of a ref class.

*reference-class-declaration:*
*ref-class-key* *identifier* `;`

*ref-class-key:*
`ref class`
`ref struct`

A `ref class` declaration and `ref struct` declaration differ in the default accessibility of members. The members of a `ref class` are private by default. On the other hand, the members of a `ref struct` are public by default.

A *reference-class-definition* defines a ref class.

*reference-class-definition:*
*attributes*$_{opt}$ *top-level-type-visibility*$_{opt}$ *ref-class-key* *identifier*
*class-modifiers*$_{opt}$ *base-clause*$_{opt}$ `{` *member-specification*$_{opt}$ `}` `;`

A *reference-class-definition* can include a set of *attributes* (§28), *top-level-type-visibility* (§12.4), *class-modifier*s (§18.1.1), and *base-clause* (§20.1.1).

### 20.1.1 Ref class base specification

A *reference-class-definition* can include a *base-clause* specification, which defines the direct base class of the ref class, and the interfaces implemented by the 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 assumed to be `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 *reference-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 §24.4.) Those interface types shall be at least as accessible as the ref class itself.

## 20.2 Ref class members

Add text to indicate the circumstances under which the following type modifiers shall be emitted, and point to each modifier's definition:[[#148]]

- IsConst (i.e., data member involves a cv type).
- IsImplicitlyDereferenced (i.e., has a reference type).

- IsLong (i.e., long/unsigned long/long double type).
- IsSignUnspecifiedByte (i.e., plain char's sigedness).
- IsVolatile (i.e., data member involves a cv type).

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 member function of a ref class shall not have a *cv-qualifier-seq*.

### 20.2.1 Variable initializers

The definition of ***zero-initialize*** in the C++ Standard (§8.5/5) has been extended, 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) has been extended, as follows:

"If no initializer is specified for a handle, the handle is always zero-initialized. Otherwise, if no initializer is specified for a nonstatic object, the object and its subobjects, if any, have an indeterminate initial value);"

[*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*]

Tracking references are treated like Standard C++ references—they are always initialized.

## 20.3 Functions

Add text to indicate the circumstances under which the following type modifiers shall be emitted, and point to each modifier's definition:

- IsBoxed i.e., passing a handle to a value type).
- IsByValue (i.e., ref class type passed by value).
- IsConst (i.e., pointer or reference to a const-qualified type).
- IsExplicitlyDereferenced (i.e., interior_ptr as a parameter).
- IsImplicitlyDereferenced (i.e., parameter is a reference).
- IsLong (i.e., long/unsigned long/long double parameters).
- IsExplicitlyDereferenced (i.e., pin_ptr as a parameter).
- IsSignUnspecifiedByte (i.e., plain char's signedness).
- IsUdtReturn (i.e., ref class type returned by value).
- IsVolatile (i.e., pointer or reference to a volatile-qualified type).

A virtual member function in a ref class can contain:

- the *function-modifier* `override`, or an *override-specifier*, or both (§18.3.1).
- the *function-modifier* `sealed` (§18.3.2).
- the *function-modifier* `abstract` (§18.3.3).

Virtual function overrides in ref classes shall not have covariant return types. [*Rationale:* This is a restriction imposed by the CLI. *end rationale*]

Member functions in a ref class can optionally have a *parameter-array* (§18.3.6) in their *parameter-declaration-clause*.

For each ref class, the implementation shall reserve several names (§18.2.3). A program is ill-formed if it declares a member whose name matches any of these reserved names.

## 20.4 Properties

Ref classes support properties (§18.4).

For each property definition, the implementation shall reserve several names (§18.2.1). A program is ill-formed if it declares a member whose name matches any of these reserved names.

## 20.5 Events

Ref classes support events (§18.5).

For each event definition, the implementation shall reserve several names (§18.2.2). A program is ill-formed if it declares a member whose name matches any of these reserved names.

## 20.6 Static operators

Ref classes support static operators (§18.6).

## 20.7 Instance constructors

## 20.8 Static constructor

Ref classes support static constructors (§18.8).

## 20.9 Literal fields

Ref classes support literal fields (§18.9).

## 20.10 Initonly fields

Ref classes support initonly fields (§18.10).

## 20.11 Destructors and finalizers

*See* §18.11.

## 20.12 Delegates

Ref classes support *delegate-definition*s (§26).

A ref class is permitted to contain a field having a delegate type.

# 21. Value classes

Introduce value classes -- Discuss the following: value classes are optimized for small data structures. As such, value classes do not allow inheritance from anything but interface classes. [[#66]]

[*Note:* As described in §12.2.2, the fundamental types provided by C++/CLI, such as `int`, `double`, and `bool`, are, in fact, all value classes. Just as these predefined types are value classes, it is also possible to use value classes and operator overloading to implement new "primitive" types in this specification. Two examples of such types are given at the end of this clause (§??). *end note*]

## 21.1 Value class declarations

A *value-class-declaration* introduces a declaration of a value class.

*value-class-declaration:*
    *value-class-key identifier* `;`

*value-class-key:*
    `value class`
    `value struct`

A `value class` declaration and `value struct` declaration differ in the default accessibility of members. The members of a `value class` are private by default. The members of a `value struct` are public by default.

A *value-class-definition* defines a value class.

*value-class-definition:*
    *attributes*$_{opt}$ *top-level-type-visibility*$_{opt}$ *value-class-key identifier*
        *value-class-modifier*$_{opt}$ *base-clause*$_{opt}$ `{` *member-specification*$_{opt}$ `}` `;`

A *value-class-definition* can include a set of *attributes* (§28), *top-level-type-visibility* (§12.4), *value-class-modifier* (§21.1.1), and *base-clause*(§21.1.2).

### 21.1.1 Value class modifiers

A *value-class-definition* can optionally include a modifier:

*value-class-modifier:*
    `sealed`

The `sealed` modifier is discussed in §18.1.1.2. All value classes are implicitly sealed (so the explicit use of this modifier in this context is redundant).

### 21.1.2 Value class base specification

A *value-class-definition* can include a *base-clause* specification, which defines the interfaces implemented by the value class. Can the base class System::ValueType redundantly be specified? No [[Ed.]]

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 that specify interface types, the value class is said to implement those interface types. (Interface implementations are discussed further in §24.4.) Those interface types shall be at least as accessible as the value class itself.

## 21.2 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*.

Except for the differences noted in §21.3, the descriptions of class members provided in §20.2 through §20.10, and §20.12 apply to value class members as well.

## 21.3 Ref class and value class differences

To be added. [[Ed]]

## 21.4 Simple value classes

Is this subclause intended to do the same thing as §12.2.2.1? If so, which one shall we keep? [[Ed]]

### 21.4.1 Constructors

Add words about instance constructors and static constructor.[[#150]]

Value classes cannot have SMFs (specifically, default constructor, copy constructor, assignment operator, destructor, or finalizer. Need to add specification for this along with rationale. [[#67]]

# 22. Mixed classes

This clause is reserved for possible future use. Consider writing text for here. [[#68]]

# 23. 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.

An Array in C++/CLI 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 Array element. The rank of an Array is also referred to as the ***dimensions*** of the Array. An Array with a rank of one is called a ***single-dimensional Array***, and an Array with a rank greater than one is called a ***multi-dimensional Array***.

Throughout this Standard, the term ***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 an 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 Array, but, rather, are established when an instance of the 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 an Array is the product of the lengths of each dimension in the Array. If one or more of the dimensions of an Array have a length of zero, the Array is said to be empty.

The element type of an Array can be any type, including an Array type.

## 23.1 Array types

An Array type is declared using a pseudo-template ref class with the following declaration:

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

The class is a pseudo-template because aspects of an Array type cannot be implemented in a library using the facilities of the language. An *array-type* is any specialization of the `cli::array` pseudo-template class. For example:

```
array<int>^ arr1D = gcnew array<int>(10);
array<int, 3>^ arr3D = gcnew array<int, 3>(10, 20, 30);
```

### 23.1.1 The System::Array type

The `System::Array` type is the abstract base type of all Array types. An implicit reference conversion (§??) exists from any Array type to `System::Array`, and an explicit reference conversion (§??) exists from `System::Array` to any Array type. Note that `System::Array` is not itself an array-type. Rather, it is a *reference-class-type* from which all *array-type* are derived.

Is reference conversion the correct term? [[#118]]

## 23.2 Array creation

Array instances are created by *array-creation-expressions* (§??) or by field or local variable declarations that include an *array-initializer* (§23.6).

When an Array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing Array instance, nor is it possible to resize its dimensions.

An Array instance created by an *array-creation-expression* is always of an Array type. The `System::Array` type is an abstract type, so it cannot be instantiated.

Elements of Arrays created by *array-creation-expression*s are always initialized to their default value (§??).

## 23.3 Array element access

Array elements are accessed using *element-access* expressions (§??) of the form `A[I1, I2, …, IN]`, where A is an expression having an Array type, and each `IX` is an expression of integral type or a type that can be implicitly converted to an integral type.

An *element-access* expression differs from subscript expressions in Standard C++ (§5.2.1) in that in the former case, commas are not treated as operators. Rather, commas separate individual expressions that respectively match the dimension of the Array being accessed. However, parentheses can be used to force the use of the comma operator in an expression. The result of an Array *element-access* is a variable, namely the Array element selected by the indices. Add examples. [[Ed]]

The elements of an Array can be enumerated using a `for each` statement (§16.2.1).

## 23.4 Array members

Every Array type inherits the members declared by the type `System::Array`. In addition, Arrays have iterators compatible with Standard C++'s template library.

Provide details for Array members. [[#73]]

## 23.5 Array covariance

For any two types `A` and `B`, if an implicit reference conversion (§??) or explicit reference conversion (§??) exists from `A` to `B`, then the same reference conversion also exists from the Array type `array<A, R>` to the Array type `array<B, R>`, where `R` is any given rank-specifier (but is the same for both Array types). This relationship is known as ***array covariance***. In particular, Array covariance means that a value of an Array type `array<A, R>` might actually be a reference to an instance of an Array type `array<B, R>`, provided an implicit reference conversion exists from `B` to `A`.

Because of Array covariance, assignments to Arrays where the elements are ref classes will include a run-time check, which ensures that the value being assigned to the Array element is actually of a permitted type (§??).

Array covariance does not extend to boxing conversions. For example, no conversion exists that permits an `array<int>` to be treated as an `array<Object^>` or `array<int^>`.

Array covariance really only applies to handles of Arrays, not direct Arrays – in other words, do Arrays have copy constructors? [[#74]]

## 23.6 Array initializers

To be added. [[#76]]

# 24. Interfaces

An interface defines a set of virtual members that an implementing class must define. An interface can also require an implementing class to implement other interfaces. A class can implement multiple interfaces.

The interface does not provide a definition for any of its members. Instead, classes that implement the interface supply these definitions.

## 24.1 Interface declarations

An *interface-class-declaration* introduces a declaration of an interface.

*interface-class-declaration:*
*interface-class-key* *identifier* `;`

*interface-class-key:*
`interface class`
`interface struct`

An `interface class` and `interface struct` declaration are equivalent. The default accessibility of members within an interface is public, and the accessibility cannot be changed.

An *interface-class-definition* defines an interface.

*interface-class-definition:*
*attributes*$_{opt}$ *top-level-type-visibility*$_{opt}$ *interface-class-key* *identifier*
*interface-class-bases*$_{opt}$ `{` *member-specification*$_{opt}$ `}` `;`

An *interface-class-definition* can include a set of *attributes* (§28), *top-level-type-visibility* (§12.4), and *interface-class-bases* (§24.1.1).

### 24.1.1 Interface base specification

An *interface-class-definition* can include an *interface-class-bases* specification, which defines the ***explicit base interfaces*** of the interface being defined.

*interface-class-bases:*
`:` *interface-class-base-list*

*interface-class-base-list:*
`public`$_{opt}$ *interface-type*
*interface-class-base-list* `,` `public`$_{opt}$ *interface-type*

The explicit base interfaces of an interface shall be at least as accessible as the interface itself (§??). [*Note*: A program is ill-formed if it specifies a `private` interface in the *interface-class-base-list* of a `public` interface. *end note*]

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.

## 24.2 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 instance functions, instance properties, instance events, or nested types of any kind. An interface cannot contain fields, operators, constructors, destructors, finalizers, or static members of any kind.

All interface members have public access. pickup the restrictions from page 333

All members declared in an interface are implicitly abstract. However, those members can redundantly contain the `virtual` and/or `abstract` modifiers, and/or a *pure-specifier*. [*Example:*

```
interface class I {
   property int Size { /*…*/ };  // (implicit) abstract property
   virtual property string Name abstract = 0 { /*…*/ };
                                 // “virtual”, “abstract” and “= 0”
                                 // permitted but are redundant
};
```

*end example*]

### 24.2.1 Interface functions

A function in an interface is declared exactly the same way as a function in a class. An interface function declaration is not permitted to specify a function definition; therefore, the declaration always ends with a semicolon.

If the function is declared `virtual`, it shall also be declared `abstract`, and vice versa.

Member functions in an interface class can optionally have a *parameter-array* (§18.3.6) in their *parameter-declaration-clause*.

For each interface class, the implementation shall reserve several names (§18.2.3). A program is ill-formed if it declares a member whose name matches any of these reserved names.

### 24.2.2 Interface properties

Interface classes support properties (§18.4).

The accessor functions of an interface property definition correspond to the accessor functions of a class property definition (§18.4.2), except that in an interface the accessor functions must 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 j] { 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 (§18.4.4). Such a 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.

For each property definition, the implementation must reserve several names (§18.2.1). A program is ill-formed if it declares a member whose name matches any of these reserved names.

### 24.2.3 Interface events

Interface classes support events (§18.5).

The accessor functions of an interface event declaration correspond to the accessor functions of a class event definition (§18.5.2), except that the accessor functions must 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.

For each event definition, the implementation must reserve several names (§18.2.2). A program is ill-formed if it declares a member whose name matches any of these reserved names.

### 24.2.4 Delegates

Interface classes support *delegate-definition*s (§26).

### 24.2.5 Interface member access

Do we need this subclause? [[Ed]]

## 24.3 Fully qualified interface member names

## 24.4 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:
   Object^ Clone() {…}
   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 implements an interface also implicitly implements all of the interface's base interfaces. This is true even if the class doesn't 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:
   void Paint() {…}
   void SetText(String^ text) {…}
};
```

Here, class `TextBox` implements both `IControl` and `ITextBox`. *end example*]

Address what happens when a ref class does not implement an interface function (and what happens when a base class has a non-virtual function with the same name). [[#76]]

# 25. Enums

An enum type is a distinct type with named constants. C++/CLI includes two kinds of enum types: ***native enums*** that are compatible with Standard C++ enums (§7.2), and ***CLI enums***, which are new, and that are preferred for frameworks programming. Native and CLI enum types are collectively referred to as ***enum types***. A native enum can only be generated by a C++ compiler. To languages other than C++, a native enum and a CLI enum appear to be exactly the same; they both cause the same metadata to be generated, and they both inherit from `System::Enum` (§25.3).

[*Example:* The example

```
public enum Suit : short { Hearts = 1, Spades, Clubs, Diamonds};
```

defines a publicly accessible 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 example

```
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*]

## 25.1 Native enums

A native enum is an enum type.

Enumerations as defined by the C++ Standard (§7.2) continue to have exactly the same meaning. Native enums have extensions to allow the following: declaration of the underlying type, the placement of attributes on enumerators, and access to enumerators within the scope of the *enum-name*.

### 25.1.1 Native enum declarations

The *enum-specifier* production in the C++ standard (§7.2) has been extended, as follows:

*enum-specifier:*
*attributes*$_{opt}$ *top-level-type-visibility*$_{opt}$ `enum` *identifier*$_{opt}$ *enum-base*$_{opt}$ `{` *enumerator-list*$_{opt}$ `}`

An *enum-specifier* can optionally include a set of *attributes* (§28), *top-level-type-visibility* (§12.4), *enum-base* (§25.1.3), and *enumerator-list*.

### 25.1.2 Native enum visibility

A non-nested native enum can optionally specify the accessibility of the native enum by using a *top-level-type-visibility* of `public` or `private` (§12.4).

### 25.1.3 Native enum underlying type

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. Unlike Standard C++, C++/CLI allows that underlying type to be specified.

*enum-base:*
`:` *??-type[[#152]]*

The underlying type of a native enum can be explicitly declared via *enum-base*, as one of the following types: `bool`, `char`, `unsigned char`, `signed char`, `short`, `unsigned short`, `int`, `unsigned int`,

long long, unsigned long long, `float`, or `double`. `wchar_t` cannot be used as an underlying type. If no underlying type is given for a native enum, the rules specified in the C++ Standard (§7.2) apply.

### 25.1.4 Native enum members

The *enumerator* production in the C++ Standard (§7.2) has been extended, as follows:

*enumerator:*
*attributes*$_{opt}$ *identifier*

The values assigned to enumerators are either explicit or implicit, as defined by the C++ Standard when the underlying type is an integral value.

## 25.2 CLI enums

A CLI enum is an enum type. All enumerations generated by CLI-based languages other than C++ are CLI enums. CLI enums are different from native enums in 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) do not apply to a CLI enum.

### 25.2.1 CLI enum declarations

A *cli-enum-declaration* introduces a declaration of a CLI enum type.

*cli-enum-declaration:*
*cli-enum-class-key* *identifier* `;`

*cli-enum-class-key:*
`enum class`
`enum struct`

An `enum class` and `enum struct` declaration are equivalent.

A *cli-enum-definition* defines a CLI enum.

*cli-enum-definition:*
*attributes*$_{opt}$ *top-level-type-visibility*$_{opt}$ *cli-enum-class-key* *identifier* *enum-base*$_{opt}$
`{` *enumerator-list*$_{opt}$ `}` `;`

A *cli-enum-definition* can optionally include a set of *attributes* (§28), *top-level-type-visibility* (§12.4), *cli-enum-class-key*, *enum-base* (§25.1.3), and *enumerator-list*.

### 25.2.2 CLI enum visibility

A non-nested CLI enum can optionally specify the accessibility of the CLI enum by using a *top-level-type-visibility* of `public` or `private` (§12.4).

### 25.2.3 CLI enum underlying type

A CLI enum can explicitly declare an underlying type, following the same rules for explicit underlying type as native enums (§25.1.3). A CLI enum definition that does not explicitly declare an underlying type has an underlying type of `int`.

### 25.2.4 CLI enum members

*See* §25.1.1.

### 25.2.5 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 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 an enum can be cast to the enum type, and is a distinct valid value of that 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`. Some members in this set require an underlying integral type.

## 25.3 The System::Enum type

The type `System::Enum` is the abstract base class of both native and CLI enum types (this is distinct and different from the underlying type of the enum type), and the members inherited from `System::Enum` are available in any enum type. A boxing conversion (§??) exists from any enum type to `System::Enum`, and an unboxing conversion (§??) exists from `System::Enum` to any enum type.

Note that `System::Enum` is not itself an enum type; it is a value class type from which all enum types are derived. The type `System::Enum` inherits from the type `System::ValueType`, which, in turn, inherits from `System::Object`.

# 26. 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 (§26.1) with the delegate's type. This makes delegates suited for "anonymous" invocation. *end note*]

## 26.1 Delegate definitions

A *delegate-definition* is a type-declaration[[Ed.]] (§??) that defines a new delegate type.

*delegate-definition:*
*attributes*$_{opt}$ *top-level-type-visibility*$_{opt}$ `delegate` *decl-specifier-seq*$_{opt}$ *identifier*
`(` *decl-specifier-seq* `)` `;`

Redo this grammar. [[#78]]

A *delegate-definition* can include a set of *attributes* (§28).

The return type of each of the functions that can be encapsulated by the delegate is indicated by *return-type*.

A non-nested delegate can optionally specify the accessibility of the class by using a *top-level-type-visibility* of `public` or `private` (§12.4).

The delegate's type name is *identifier*.

The optional *delegate-parameter-list* specifies the parameters of the delegate, and *return-type* indicates the return type of the delegate. The parameter list of a delegate corresponds to that of a function, except that at least one parameter must be specified. [*Note:* no C-style "vararg" argument is allowed, nor is a parameter array. *end note*]

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-type*s 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-definition*. 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`. `System::Delegate` is not itself a delegate type; it is a class type from which all delegate types are derived.

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 (§26.2) 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 + (§15.8.1) and += operators (§15.18). A delegate can be removed from a combination of delegates, using the binary - (§15.8.2) and -= operators (§15.18). Delegates can be compared for equality (§15.11.2).

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. A delegate 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 a duplicate entry's being called once per occurance.

When a list of entries is removed from an invocation list, the first occurance 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*]

## 26.2 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 namespace scope function. Here del-con-arg1 shall be the address of a static member function or a 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 to a instance function. Here, del-con-arg2 shall be a reference to an Object 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 Object 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 member 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*]

## 26.3 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`.

# 27. Exceptions

To be added. (Cover unification of CLI and Standard C++ exception-handling models.) [[#79]]

## 27.1 Common exception classes

The following exceptions are thrown by certain C++/CLI operations.

| Exception Name | Description |
|---|---|
| `System::NullReferenceException` | Thrown when a null-valued handle is dereferenced. |
| `System::TypeInitializationException` | Thrown when a static constructor throws an exception, yet no catch clauses exists to catch it. |

5

# 28. Attributes

The CLI enables programmers to invent new kinds of declarative information, called ***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 (§28.1), which can have positional and named parameters (§28.1.2). Attributes are attached to entities in a C++ program using attribute specifications (§28.2), and can be retrieved at run-time as attribute instances (§28.3).

## 28.1 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*]

### 28.1.1 Attribute usage

The attribute `System::AttributeUsageAttribute` (§28.4.1) 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. See §??. *end note*]

`AttributeUsage` has a positional parameter (§28.1.2) 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 *reference-class-declaration*s and *interface-class-declaration*s 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 (§28.1.2), 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 can be specified at most 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 declaration with two uses of the `Author` attribute. *end example*]

`AttributeUsage` has another named parameter (§28.1.2), 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 `X` not having an `AttributeUsage` attribute attached to it, as in

```
ref class X : Attribute { … };
```

is equivalent to the following:

```
[AttributeUsage(AttributeTargets::All, AllowMultiple = false,
Inherited = true)] ref class X : Attribute { … };
```

### 28.1.2 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.

[*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 (§30.1.1) nor an open constructed type (§30.2.1) shall be an argument to the constructor of a custom attribute.

### 28.1.3 Attribute parameter types

The types of positional and named parameters for an attribute class are limited to the ***attribute parameter types***, which are:

- One of the following types: `bool`, `char`, `wchar_t`, `short`, `int`, `long`, `long long`, `float`, `double`, and `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.

## 28.2 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 *type-declaration*s (§??), class *member-declaration*s, struct *member-declaration*s, interface *member-declaration*s, enum *member-declaration*s, *accessor-specification* (§??), and *formal-parameter*s (§??).

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.

*global-attributes:*
    *global-attribute-sections* `;`

*global-attribute-sections:*
    *global-attribute-section*
    *global-attribute-sections* *global-attribute-section*

*global-attribute-section:*
    `[` *global-attribute-target* `:` *attribute-list* `]`

*global-attribute-target:*
    `assembly`
    `module`

*attributes:*
    *attribute-sections*

*attribute-sections:*
    *attribute-section*
    *attribute-sections* *attribute-section*

*attribute-section:*
    `[` *attribute-target-specifier*$_{opt}$ *attribute-list* `]`

*attribute-target-specifier:*
    *attribute-target* `:`

*attribute-target:*
`class`
`constructor`
`delegate`
`enum`
`event`
`field`
`interface`
`method`
`parameter`
`property`
`returnvalue`
`struct`

*attribute-list:*
*attribute* `,`$_{opt}$
*attribute* `,` *attribute-list*

*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:*
*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:* A trailing comma is allowed in a *global-attribute-section* and an *attribute-section*; this provides flexibility in adding or deleting members from the list, and simplifies machine generation of such lists. *end note*]

[*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*]

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*. When an attribute is placed at file scope, a *global-attribute-target* is required. 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 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 implementation-defined. 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 (§28.1.3).
- 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*]

## 28.3 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.

### 28.3.1 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` must 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`.

### 28.3.2 Run-time retrieval of an attribute instance

This is governed by the CLI standard (see §??).

## 28.4 Reserved attributes

A small number of attributes affect the language in some way. These attributes include:

- `System::AttributeUsageAttribute` (§28.4.1), which is used to describe the ways in which an attribute class can be used.
- `System::ObsoleteAttribute` (§28.4.2), which is used to mark a member as obsolete.

Need to document C++/CLI-specific attribute ScopelessEnumAttribute. What about HasCopySemanticsAttribute? [[Ed]]

### 28.4.1 The AttributeUsage attribute

The attribute `AttributeUsage` is used to describe the manner in which the attribute class can be used.

A ref class that is decorated with the `AttributeUsage` attribute must derive from `System::Attribute`, either directly or indirectly. Otherwise, the program is ill-formed.

The constructor for class `AttributeUsageAttribute` takes an argument of type `AttributeTargets`. This enumeration 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.
- `Struct` indicates that the attribute can be applied to a value class.
- `Method` indicates that the attribute can be applied to a function.
- [*Note:* For an example of using this attribute, see §28.1.1. *end note*]

### 28.4.2 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 issue a diagnostic if no error 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")]
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*]

## 28.5 Attributes for interoperation

### 28.5.1 Interoperation with other CLI-based languages

#### 28.5.1.1 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.

Check this name; this attribute might have been renamed in the CLI standard. [[#119]]

#### 28.5.1.2 The MethodImplOption attribute

Synchronized function for compiler-generated add/remove event accessor functions. [[#113]]

# 29. Templates

This clause is incomplete.[[#82]]

The template syntax is the same for all types, including CLI types. Templates on CLI 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.

## 29.1 Attributes

Given that the grammars for ref class, value class, and interface class already include the possibility of attributes, review what is stated below and modify as necessary. (Support for attributes has yet to be added to the grammar for functions.) [[#82]]

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>
[attributes]
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>
[attributes]
void f(const T& t) { /* … */ }
```

*end example*]

Explicit and partial specializations of a class template must have the same class kind as the primary template. For example, an explicit specialization of a ref class template cannot be a value class. [[#82]]

Are there any issues with metadata name emission? Is it even necessary to standardize this since template specializations are really only useful inside an assembly. [[#82]]

## 29.2 Type deduction

There is no ordering among %, ^, &, or *.

Template type deduction of `nullptr` literal is not possible.

Non-type template parameters will not include %, ^, or nullptr. [[#82]]

# 30. Generics

Some issues to consider are: (1) using templates inside of generics, (2) overloading rules, and (3) dynamic cast to type parameters. The high level goal with generics (as with other parts of C++/CLI) is to provide a close mapping of the underlying capabilities of the CLI, which means that C++ can potentially create generics that other languages might not be able to consume. Not all languages support all capabilities, but C++/CLI supports more than most. (However, C++/CLI does not support array co- or contra-variance.)[[#98]]

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 must 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***.

## 30.1 Generic declarations

To accommodate the addition of generics, the grammar for *declaration* in the C++ Standard (§7) has been extended, as follows:

*declaration:*
…
*generic-declaration*

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 (§30.1.1). Constraints are defined via a *constraint-clause-list* (§30.4).

If the *declaration* of a *generic-declaration* is other than a ref class, value class, interface class, delegate, or function (excluding constructors and destructors), the program is ill-formed.

A program is ill-formed if it declares a property or event as a generic. The constituent 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 as a namespace scope or class scope declaration.

The text indicates that a generic-declaration may appear in a class scope, but the syntax of member-declaration has not been extended to permit a generic-declaration. [[#153]]

Generic declarations that are also definitions can have public or private assembly visibility (§10.2.1), except that a non-member function definition shall never have public visibility.

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. Doesn't the text "a generic name declared in namespace scope or in class scope shall be unique in that scope" make the first sentence of this paragraph redundant? Re the reference to 14.5.4: That is the section on partial specialization. Generics can't be partially specialized, can they? The spec. should probably answer that explicitly. [[#154]]

Generic type declarations follow the same rules as non-generic type declarations except where noted. What is a non-generic type? Does it mean that the rules are the same as classes? As template classes? Something else? [[#155]] Generic type declarations can be nested inside non-generic type declarations. Can generic types be nested in native classes? [[#156]]

Generic functions are discussed further in (§30.3).

Type Overloading – This involves overloading on arity, and is currently under investigation. Such a feature permits the following: [[#157]]

```
ref class X {};
generic<typename T>
ref class X {};
generic<typename T, typename U>
ref class X {};
```

### 30.1.1 Type parameters

A type parameter can be defined in one of the following ways:

*generic-parameter:*
*attributes*$_{opt}$ `class` *identifier*
*attribute*$_{opt}$ `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 (§28).

A *generic-parameter* defines its *identifier* to be a *type-name*. The equivalent wording for template parameters in the working paper has been changed to "defines its identifier to be a typedef-name". The revised wording should probably be used here too (see core issue 283) [[#158]].

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 (§30.2). 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.

### 30.1.2 Referencing a generic type by name

Like templates in Standard C++, within the body of a generic type any usage of the unqualified unadorned name of that type is assumed to refer to the current instantiation. 30.1.3 describes "The instance type". These seem like two different ways of describing the same concept. Can they be unified in some way?[[#159]] [*Example*:

```
generic<typename T>
ref class X {
public:
   X() {}      // ok: means X<T>
   void f(X^); // ok: means X<T>
   ::X g();    // error
};
```

*end example*]

Outside its declaration, a generic type is referenced using a constructed type (§30.2). [*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 (§30.2.1). A constructed type that uses no type parameters, such as `List<int>`, is called a closed constructed type (§30.2.1). *end example*]

### 30.1.3 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 (§30.2) 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 const-qualified handle to the instance type. Inside the declaration of a value class, `this` is a const-quafied `interior_ptr` to the instance type. For non-generic types, the instance type is simply the declared type. [*Example*: The following shows several class declarations along with their instance types:

```
generic<typename T>
ref class A {        // instance type: A<T>
   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*]

### 30.1.4 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 declaration shall not use `System::Attribute` as a direct or indirect base class.

### 30.1.5 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 (§30.1.2) 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, and destructors 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 Array, the destructor of a generic class will not invoke the destructor on such a member.

Within a generic class declaration, access to inherited protected instance members is permitted through an instance of any 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();
```

```
            D<int>^ di = gcnew D<int>;
            di->x = 123;

            D<String^>^ ds = gcnew D<String^>;
            ds->x = "test";
        }
    };
```

the three assignments to x are permitted because they all take place through instances of class types constructed from the generic type. *end example*]

Static operators are discussed in (§30.1.7), other static members are discussed in (§30.1.6), nested types are discussed in (§30.1.10), and generic functions, in general, are discussed in (§30.3).

### 30.1.6 Static members

This subclause describes when a static constructor is invoked. In 18.8, it references the CLI Standard Partition II (10.5.3). Are the rules the same? Should this subclause also just reference the CLI spec?[[#160]]

A static data member in a generic class declaration is shared amongst all instances of the same closed constructed type (§30.1.2), 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 declaration. 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(typeid<C<T> >);
   }

   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[System.Int32]
1
C[System.Double]
1
1
1
2
```

*end example*]
Static operators are discussed in (§30.1.7)

### 30.1.7 Operators

Generic class declarations can define operators, following the same rules as non-generic class declarations. The instance type (§30.1.3) of the class declaration must be used in the declaration of operators in a manner analogous to the normal rules for operators, as follows:

- A unary operator shall take a single parameter of a handle to the instance type.
- The unary ++ and -- operators shall take a single parameter of a handle to the instance type and return a handle to the same type.
- At least one of the parameters of a binary operator shall be a handle to the instance type.

[*Example*: The following shows some examples of valid operator declarations in a generic class:

```
generic<typename T>
public ref class Vector {
public:
   Vector(int size) { … };
   static Vector<T>^ operator-(Vector<T>^ v) { … }
   static Vector<T>^ operator++(Vector<T>^ v) { … }
   static Vector<T>^ operator+(Vector<T>^ v1, Vector<T>^ v2) { … }
   // …
};

int main() {
   Vector<int>^ iv1 = gcnew Vector<int>(5);
   Vector<int>^ iv2;

   iv2 = iv1++;
   iv2 = ++iv1 + -iv1;
}
```

*end example*]
What to say about explicit conversion functions (which can only occur in managed class types)?[[#161]]

### 30.1.8 Member overloading

Functions, instance constructors, and static operators within a generic class declaration 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.

### 30.1.9 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 §30.2.4. 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:
   String^ F() override { … }        // Ok
   C<String^>^ G() override { … }    // Ok
   void H(C<int>^ x) override { … } // Error, should be C<String^>
};

generic<typename T, typename U>
ref class E : C<U> {
public:
   U F() override { … }              // Ok
   C<U>^ G() override { … }          // Ok
   void H(C<T>^ x) override { … }    // Error, should be C<U>
};
```

*end example*]

### 30.1.10 Nested types

A generic class declaration can contain nested type declarations, except that a generic class declaration 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 declaration 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, must 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 class 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*]

## 30.2 Constructed types

A generic type declaration, by itself, does not denote a type. Instead, a generic type declaration is used as a blueprint to form many different types, by way of applying type arguments (§30.2.1). 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 (§30.2.1)

To accommodate the addition of generics, the grammar for *unqualified-id* in the C++ Standard (§5.1) has been extended, as follows:

*unqualified-id:*
…
*generic-id*

A constructed type is referred to by a *generic-id*:

*generic-id:*
*generic-name* < *generic-argument-list* >

*generic-name:*
*identifier*

*generic-argument-list* is discussed in (§30.2.2).

### 30.2.1 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.
- An 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 (§30.2.2) 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 are, 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.

### 30.2.2 Type arguments

This subclause lists the types that can and cannot be generic arguments. Fundamental types are not included in either set, neither are function types. The subclause does not say whether or not cv-qualified types are allowed.[[#162]]

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 (§30.3) 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 an 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, or a ref class by value as a generic argument. end note]

Each *generic-argument* shall satisfy any constraints (§30.4) on the corresponding type parameter.

### 30.2.3 Base classes and interfaces

A constructed class type has a direct base class. If the generic class declaration does not specify a base class, the base class is `System::Object`. If a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *generic-parameter* in the base class declaration, the corresponding *generic-argument* of the constructed type. [*Example*: Given the generic class declarations

```
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 declarations on the generic type declaration, and substituting, for each *generic-parameter* in the base interface declaration, 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 declarations:

```
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*]

### 30.2.4 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. It would be helpful to explain this in more detail and/or give an example where this makes a difference.

[*Example*: Given the generic class declaration

```
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 declaration `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*]

### 30.2.5 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.

More precisely, the accessibility domain for a constructed type is the intersection of the accessibility domains of the open type and its type arguments.

## 30.3 Generic functions

Can a generic function be declared inside a native class? (No) Can generic functions (and member functions of generic classes, for that matter) have exception specifications? (No) If so, can they refer to open constructed types?[[#164]]

Member functions and non-member functions can be declared generic (§30.1). When a generic function is declared inside a ref class, value class, or interface declaration, 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. Are the nondeduced context rules the same as Standard C++ or not? The sentence before this is true, but not complete if the rules are the same as Standard C++.[[#165]]

What, if anything, does it mean for a generic function to be static/extern or inline?[[#166]

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, reference, or handle declarators. What about cv-qualifiers?[[#167]] 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.

Can you take the address of a generic function instance?[[#168]]

### 30.3.1 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 must 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*]

### 30.3.2 Type deduction

This subclause uses both the terms "type deduction" and "type inference". "Type deduction" should be used uniformly.[[#Ed.]]

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 inference allows a more convenient syntax to be used for calling a generic function, and allows the programmer to avoid specifying redundant type information.

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 (§18.3.6). The issue raised in 8.15.3 is somewhat answered here. 18.3.6 seems to deal with expanded forms of calls, not expanded forms of function declarations. I interpret the text above as saying that deduction is done as if the function were declared like this:

```
generic <typename ItemType>
void PushMultiple(Stack<ItemType>^, ItemType i1, ItemType i2,/* ... */);
```

Is that correct? I think this requires a more detailed description.[[#169]]

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 can be given explicitly or be determined by type deduction. 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.

Something needs to be said about instantiating a generic delegate using a generic function.[[#170]]

## 30.4 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*
*constraint-clause-list* *constraint-clause*

*constraint-clause:*
`where` *identifier* `:` *constraint-item-list*

*constaint-item-list:*
*constraint-item*
*constraint-item-list* `,` *constraint-item*

*constraint-item:*
*type-id*

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.

A *constraint-item-list* can include any of the following *constraint-item*s, in any order: a single class constraint and one or more interface constraints (with each being specified via a *type-id*).

If a *constraint-item* is a class type or an interface type, that type specifies a minimal "base type" that every type argument used for that type parameter shall support. Whenever a constructed type or generic function is used, the type argument is checked against the constraints on the type parameter at compile-time. The type argument supplied shall derive from or implement all of the constraints given for that type parameter.

The type specified by *type-id* in a class constraint shall be a ref class type that is not `sealed`, and that type shall not be any of the following: `System::Array`, `System::Delegate`, `System::Enum`, or `System::ValueType`. A *constraint-item-list* shall contain no more than one constraint that is a class type.

The type specified by *type-id* in an interface constraint shall be an interface class type. The same interface type shall not be specified more than once in a given *constraint-clause*.

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, but the constraint shall not be a type parameter alone.

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.

[*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*]

### 30.4.1 Satisfying constraints

Whenever a constructed type is used or a generic function is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or function. For each *constraint-clause*, the type argument `A` that corresponds to the named type parameter is checked against each constraint as follows. 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 must be the case that type `A` is convertible to type `C` by one of the following:

- An identity conversion (§??)
- An implicit reference conversion (§??)
- A boxing conversion (§14.4)
- An implicit conversion from a type parameter `A` to `C` (§??).

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` must 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*]

### 30.4.2 Member lookup on type parameters

The results of member lookup in a type given by a type parameter `T` depends on the constraints, if any, specified for `T`. If `T` has no constraints, then member lookup on `T` returns the same set of members as member lookup on `System::Object`. Otherwise, the first stage of member lookup considers all the members in each of the types that are constraints for `T`. After performing the first stage of member lookup for each of the type constraints of `T`, the results are combined, and then hidden members are removed from the combined results. When are members considered hidden? Is it using the rules described later? Those are described as applying only when a type parameter has both a class constraint and one or more interface constraints though.[[#171]]

When a type parameter has both a class constraint and one or more interface constraints, member lookup can return a set of members, some of which were declared in the class, and others of which were declared in an interface. The following additional rules handle this case.

- During member lookup, members declared in a class other than `System::Object` hide members declared in interfaces.
- During overload resolution of functions, if any applicable member was declared in a class other than `System::Object`, all members declared in an interface are removed from the set of considered members.

These rules only have effect when doing binding on a type parameter with both a class constraint and an interface constraint. Informally, members defined in a class constraint are always preferred over members in an interface constraint.

### 30.4.3 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.

### 30.4.4 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 have both class and interface constraints, type conversions defined in a class constraint are always preferred over those in an interface constraint

Miscellaneous generics issues:

1. I seem to recall discussions of other kinds of constraints (I believe one of them concerned whether you could do a "new T()").

2. Doesn't there need to be some discussion of how overload resolution works when a function argument has a type parameter as its type?

3. Are the typename and template rules for syntactic disambiguation the same in generics as in templates? Presumably, the lack of specialization would eliminate the need for these.

4. If scope contains a set of overloaded generic functions, is partial ordering used to choose between them?

5. I assume since there is nothing that says otherwise, that generics can be friends of other classes and generics can make other classes, functions, (including generics) friends?

6. If friendship is supported, can a generic first be declared in a friend declaration (suggested answer: no).

7. Standard C++ has restrictions on type parameters such as prohibiting types with no linkage. Does this rule apply to generic arguments?

8. Are there generic conversion functions?[[#172]]

# 31. Standard C and C++ libraries

Describe synchronization of standard C++ streams and System::Console. [[#7]]

What else should go here? [[#84]]

# 32. CLI libraries

## 32.1 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 §32.1.1. Custom modifiers have no other effect on the operation of the VES.

### 32.1.1 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([a]n.IsConst)* p2) … { … }
.field private static int32* p3
.field private static int32 modopt([a]n.IsConst)* p4
```

where a designates the parent assembly of the `IsConst` type, while n designates that type's namespace. (These can vary by modifier, and are provided as part of each modifier's specification (§32.1.5).) [*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 = 0;
X::F(q1);
//    call void X::F(int32*)

const int* q2 = 0;
X::F(q2);
//    call void X::F(int32 modopt([a]n.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;
}
// code generated:
.method public static void F(int32* p1) … {
  …
  ldarg.0
  stsfld int32* X::p3
  ldarg.0
  stsfld int32 modopt([a]n.IsConst)* X::p4
  …
}

static void F(const int* p2) {
   p4 = p2;
}
// code generated:
.method public static void F(int32 modopt([a]n.IsConst)* p2) … {
  …
  ldarg.0
  stsfld int32 modopt([a]n.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.)

### 32.1.2 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 should not be ignored, while an optional modifier can simply be ignored. For example, `volatile`-qualified data members must be marked with the `IsVolatile` modreq. The presence of this modifier cannot be ignored, as all dereferences of such members must involve the use of the `volatile.` prefixed instruction (see §32.1.5.10 for an example). On the other hand, the `const` qualifier can be modelled with a modopt since a `const`-qualified data member or 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.

### 32.1.3 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 ref class having public accessibility. (Typically, a modifier class is sealed and has no public members.) [*Example*: Here are data and function member definitions, and the metadata produced for each of their declarations:

```
public ref class X {
   int f1;
   //    .field private int32 f1

   const int f2;
   //    .field private int32 modopt([a]n.IsConst) f2

   const int* f3;
   //    .field private int32 modopt([a]n.IsConst)* f3

   const int** f4;
   //    .field private int32 modopt([a]n.IsConst)** f4

   const int* const* f5;
   //    .field private int32 modopt([a]n.IsConst)*
   //    modopt([a]n.IsConst)* f5

   array<int>^ f6;
   //    .field private int32[] f6

   array<int*>^ f7;
   //    .field private int32*[] f7

   const array<int>^ f8;
   //    .field private int32[] modopt([a]n.IsConst) f8

   array<const int>^ f9;
   //    .field private int32 modopt([a]n.IsConst)[] f9

   const int* F() { … }
   //    .method private instance int32 modopt([a]n.IsConst)* F() … { … }

   void F(int x, const int*y, array<int>^ z) { … }
   //    .method private instance void F(int32 x,
   //       int32 modopt([a]n.IsConst)* y, int32[] z) … { … }
};
```

*end example*]

### 32.1.4 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([a1]n1.IsVolatile)
//    modopt([a2]n2.IsConst) m
```

*end example]*

To ensure that signatures for the same *Type* produced by different implementations match, the ordering in such a set of modreqs and modopts is as follows: first modreqs in ascending order by name, then modopts in ascending order by name, with case being significant. [We need some rule here; is this the one?][[#173]].

If IsBoxed is retained for the standard, we have an ordering issue to consider: Currently, the value-type special modopt is emitted before the IsBoxed modreq. For example, class [mscorlib]System.ValueType modopt([mscorlib]System.Int32) modreq([a]n.IsBoxed). That puts a modopt before a modreq. [[#174]]

### 32.1.5 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++-specific.

#### 32.1.5.1 IsBoxed

This modifier is a workaround for the MS implementation. Does it have any long-term value for the standard, even if only as an historical note?[[#175]]

This type supports the handle type punctuator ^ when used with value types.

Modreq or modopt: modreq

Assembly: ??

Namespace: ??

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 parameters that are handles to value types. [*Example*:

```
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) { … }
};

// code generated:
.field private int32* m1

.field private class [mscorlib]System.ValueType
   modopt([mscorlib]System.Int32) modreq([a]n.IsBoxed) m2

.field private class [mscorlib]System.ValueType modopt(V)
   modreq([a]n.IsBoxed) m3

.field private class C m4

// code generated:
.method public instance void F(int32* x) … { … }

.method public instance void F(class [mscorlib]System.ValueType
   modopt([mscorlib]System.Int32) modreq([a]n.IsBoxed) x) … { … }

.method public instance class [mscorlib]System.ValueType
   modopt([a]n.IsConst) modopt([mscorlib]System.SByte)
   modreq([a]n.IsBoxed) F(class [mscorlib]System.ValueType modopt(V)
   modreq([a]n.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.

The `IsBoxed` modopt is what indicates that the signature involves a handle to a boxed value type. *end example*]

#### 32.1.5.2 IsByValue

This type supports the passing of objects of a ref class type by value.

Modreq or modopt: modreq

Assembly: ??

Namespace: ??

Description:

This type is used in the signature of a function. This modreq is not used to indicate that a ref class value is returned by a function; for that, see `IsUdtReturn` (§32.1.5.9). [*Example*: Pending *end example*]

#### 32.1.5.3 IsConst

This type supports the `const` qualifier.

Modreq or modopt: modopt

Assembly: ??

Namespace: ??

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 §32.1.1, §32.1.3, and §32.1.4.

#### 32.1.5.4 IsExplicitlyDereferenced

This type supports the use of the type `interior_ptr` as a parameter.

Modreq or modopt: modopt

Assembly: ??

Namespace: ??

Description:

This type is used in the signature of any function. [*Example*:

```
public ref class X {
public:
   void F(interior_ptr<int> x) { … }
   void F(interior_ptr<unsigned char> x) { … }
};

// code generated:
.method public instance void F(int32&
   modopt([a]n.IsExplicitlyDereferenced) x) … { … }
.method public instance void F(uint8&
   modopt([a]n.IsExplicitlyDereferenced) x) … { … }
```

*end example*]

#### 32.1.5.5 IsImplicitlyDereferenced

This type is supports the reference type punctuator &.

Modreq or modopt: modopt

Assembly: ??

Namespace: ??

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. [*Example*:

```
public ref class X {
   int* m1;
   int& m2;
public:
   void F(int* x) { … }
   void F(int& x) { … }
};

// code generated:
.field private int32* m1
.field private int32* modopt([a]n.IsImplicitlyDereferenced) m2
.method public instance void F(int32* x) … { … }
.method public instance void F(int32*
   modopt([a]n. IsImplicitlyDereferenced) x) … { … }
```

*end example*]

#### 32.1.5.6 IsLong

As to whether or not this standard will map long, unsigned long, and long double to CLI types, is yet to be determined. However, if any/all of them are, here's how this modifier would be used.[[Ed.]]

This type is used 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`.

Modreq or modopt: modopt

Assembly: ??

Namespace: ??

Description:

`IsLong` can be used in the signature of any data member or function. [*Example*:

```
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) { … }
};

// code generated:
.field private int32 i
.field private int32 modopt([a]n.IsLong) li
.field private float64 d
.field private float64 modopt([a]n.IsLong) ld
.method public instance uint32 F(uint32* pu) … { … }
```

```
.method public instance uint32 modopt([a]n.IsLong)
   F(uint32 modopt([a]n.IsLong)* pul) … { … }
.method public instance float64 F(float64* pd) … { … }
.method public instance float64 modopt([a]n.IsLong)
   F(float64 modopt([a]n.IsLong)* pld) … { … }
```

*end example*]

#### 32.1.5.7 IsPinned

This type supports the use of the type `pin_ptr` as a parameter.

Modreq or modopt: modopt

Assembly: ??

Namespace: ??

Description:

This type is used in the signature of any function. [*Example*:

```
public ref class X {
public:
   void F(pin_ptr<int> x) { … }     // won't compile, yet[[Ed.]]
};
// code generated:
…
```

*end example*]

#### 32.1.5.8 IsSignUnspecifiedByte

This type supports plain `char`'s being a type separate from `signed char` and `unsigned char`.

Modreq or modopt: modopt

Assembly: ??

Namespace: ??

Description:

`IsSignUnspecifiedByte` 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([a]n.IsSignUnspecifiedByte) c
.field private int8 sc
.field private uint8 uc
.method public instance int8 modopt([a]n.IsSignUnspecifiedByte)*
   F(int8 modopt([a]n.IsSignUnspecifiedByte)* p1) … { … }
.method public instance int8 modopt([a]n.IsSignUnspecifiedByte)*
   F(int8* p2) … { … }
.method public instance int8 modopt([a]n.IsSignUnspecifiedByte)*
   F(uint8* p2) … { … }
```

while that generated from an implementation in which a plain `char` is unsigned, is shown below:

```
.field private uint8 modopt([a]n.IsSignUnspecifiedByte) c
.field private int8 sc
.field private uint8 uc
.method public instance uint8 modopt([a]n.IsSignUnspecifiedByte)*
   F(uint8 modopt([a]n.IsSignUnspecifiedByte)* p1) … { … }
.method public instance uint8 modopt([a]n.IsSignUnspecifiedByte)*
   F(uint8* p2) … { … }
.method public instance uint8 modopt([a]n.IsSignUnspecifiedByte)*
   F(uint8* p2) … { … }
```

*end example*]

#### 32.1.5.9 IsUdtReturn

This type supports the returning of objects of a ref class type by value.

Modreq or modopt: modreq

Assembly: ??

Namespace: ??

Description:

This type is used in the signature of a function. This modreq is not used to indicate a ref class value is passed to a function; for that, see `IsByValue` (§32.1.5.2). [*Example*: Pending [[Ed.]] *end example*]

#### 32.1.5.10 IsVolatile

This type supports the `volatile` qualifier. (Although `IsVolatile` is part of the CLI Standard, it is documented here as well, for convenience.)

Modreq or modopt: modreq

Assembly: `mscorlib`

Namespace: `System::Runtime::CompilerServices`

Description:

This type can be used in the signature of any data member or function.

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;
   }
};

// code generated:
.field private int32
   modreq([mscorlib]System.Runtime.CompilerServices.IsVolatile)* p1
.method public instance void F(int32
   modreq([mscorlib]System.Runtime.CompilerServices.IsVolatile)* p2,
   int32* p3) cil managed {
   …
```

```
        ldarg.0
        ldfld int32 modreq([mscorlib]
            System.Runtime.CompilerServices.IsVolatile)*
            IsVolatileEx::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)* IsVolatileEx::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*]

# Annex A. Verifiable code

To be added. [[#87]]

# Annex B. Documentation comments

To be added. [[#88]]

# Annex C. Non-normative references

ISO/IEC 23270:2003, *Programming languages — C#*.

# Annex D. CLI naming guidelines

**This annex is informative.**

Add guidelines for generics. [[Ed]]

One of the most important elements of predictability and discoverability is the use of a consistent naming pattern. Many of the common user questions don't even arise once these conventions are understood and widely used. There are three elements to the naming guidelines:

1. Casing – use of the correct capitalization style
2. Mechanical – use nouns for classes, verbs for functions, etc.
3. Word choice – use consistent terms across class libraries.

The following subclause lays out rules for the first two elements, and some philosophy for the third.

## D.1 Capitalization styles

The following subclause describes different ways of capitalizing identifiers.

### D.1.1 Pascal casing

This convention capitalizes the first character of each word. For example:

```
Color    BitConverter
```

### D.1.2 Camel casing

This convention capitalizes the first character of each word except the first word. For example:

```
backgroundColor    totalValueCount
```

### D.1.3 All uppercase

Only use all uppercase letters for an identifier if it contains an abbreviation. For example:

```
System::IO
System::WinForms::UI
```

### D.1.4 Capitalization summary

The following table summarizes the capitalization style for the different kinds of identifiers:

| Type | Case | Notes |
|---|---|---|
| Class | PascalCase | |
| Class, attribute | PascalCase | Has a suffix of Attribute |
| Class, exception | PascalCase | Has a suffix of Exception |
| Literal | PascalCase | |
| Enum type | PascalCase | |
| Enum value | PascalCase | |
| Event | PascalCase | |
| Field, non-public instance | camelCase | |
| Field, public instance | PascalCase | Rarely used (use a property instead) |

| Type | Case | Notes |
|---|---|---|
| Function | PascalCase | |
| Interface | PascalCase | Has a prefix of I |
| Local variable | camelCase | |
| Namespace | PascalCase | |
| Parameter | camelCase | |
| Property | PascalCase | |

## D.2 Word choice

- Do avoid using class names duplicated in heavily used namespaces. For example, don't use the following for a class name.

```
System    Collections    Forms    UI
```

- Do not use abbreviations in identifiers.
- If you must use abbreviations, do use camelCase for any abbreviation containing more than two characters, even if this is not the usual abbreviation.

## D.3 Namespaces

The general rule for namespace naming is `CompanyName::TechnologyName`.

- Do avoid the possibility of two published namespaces having the same name, by prefixing namespace names with a company name or other well-established brand. For example, `Microsoft::Office` for the Office Automation classes provided by Microsoft.
- Do use PascalCase, and separate logical components with two colons (as in `Microsoft::Office::PowerPoint`). If your brand employs non-traditional casing, do follow the casing defined by your brand, even if it deviates from normal namespace casing (for example, `NeXT::WebObjects`, and `ee::cummings`).
- Do use plural namespace names where appropriate. For example, use `System::Collections` rather than `System::Collection`. Exceptions to this rule are brand names and abbreviations. For example, use `System::IO` not `System::IOs`.
- Do not have namespaces and classes with the same name.

## D.4 Classes

- Do name classes with nouns or noun phrases.
- Do use PascalCase.
- Do use sparingly, abbreviations in class names.
- Do not use any prefix (such as "C", for example). Where possible, avoid starting with the letter "I", since that is the recommended prefix for interface names. If you must start with that letter, make sure the second character is lowercase, as in `IdentityStore`.
- Do not use any underscores.

```
public ref class FileStream { … };
public ref class Button { … };
public ref class String { … };
```

## D.5 Interfaces

- Do name interfaces with nouns or noun phrases, or adjectives describing behavior. For example, `IComponent` (descriptive noun), `ICustomAttributeProvider` (noun phrase), and `IPersistable` (adjective).
- Do use PascalCase.
- Do use sparingly, abbreviations in interface names.
- Do not use any underscores.
- Do prefix interface names with the letter "I", to indicate that the type is an interface.
- Do use similar names when defining a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the "I" prefix in the interface name. This approach is used for the interface `IComponent` and its standard implementation, `Component`.

```
public interface class IComponent { … };
public ref class Component : IComponent { … };
public interface class IServiceProvider{ … };
public interface class IFormatable { … };
```

## D.6 Enums

- Do use PascalCase for enums.
- Do use PascalCase for enum value names.
- Do use sparingly, abbreviations in enum names.
- Do not use a family-name prefix on enum.
- Do not use any "Enum" suffix on enum types.
- Do use a singular name for enums.
- Do use a plural name for bit fields.
- Do define enumerated values using an enum if they are used in a parameter or property. This gives development tools a chance at knowing the possible values for a property or parameter.

```
public enum class FileMode
{
   Create,
   CreateNew,
   Open,
   OpenOrCreate,
   Truncate
};
```

- Do use the `Flags` custom attribute if the numeric values are meant to be bitwise ored together.

```
[Flags]
public enum class Bindings
{
   CreateInstance,
   DefaultBinding,
   ExcatBinding,
   GetField,
   GetProperty,
   IgnoreCase,
   InvokeMethod,
   NonPublic,
   OABinding,
   SetField,
   SetProperty,
   Static
};
```

- Do use `int` as the underlying type of an enum. (An exception to this rule is if the enum represents flags and there are more than 32 flags, or the enum might grow to that many flags in the future, or the type needs to be different from `int` for backward compatibility.)
- Do use enums only if the value can be completely expressed as a set of bit flags. Do not use enums for open sets (such as operating system version).

## D.7 Static members

- Do name static members with nouns, noun phrases, or abbreviations for nouns.
- Do name static members using PascalCase.
- Do not use Hungarian-type prefixes on static member names.

## D.8 Parameters

- Do use descriptive names such that a parameter's name and type clearly imply its meaning.
- Do name parameters using camelCase.
- Do prefer names based on a parameter's meaning, to names based on the parameter's type. It is likely that development tools will provide the information about type in a convenient way, so the parameter name can be put to better use describing semantics rather than type.
- Do not reserve parameters for future use. If more data is need in the next version, a new overload can be added.
- Do not use Hungarian-type prefixes.

```
Type GetType(String^ typeName)
string Format(String^ format, array<Object^>^ args)
```

## D.9 Functions

- Do name functions with verbs or verb phrases.
- Do name functions with PascalCase.

```
RemoveAll()    GetCharArray()    Invoke()
```

## D.10 Properties

- Do name properties using noun or noun phrases.
- Do name properties with PascalCase.

## D.11 Events

- Do name event handlers with the `EventHandler` suffix.

```
public delegate void MouseEventHandler(Object^ sender, MouseEvent^ e);
```

- Do use two parameters named `sender` and `e`. The sender parameter represents the Object that raised the event, and this parameter is always of type `Object`, even if it is possible to employ a more specific type. The state associated with the event is encapsulated in an instance `e` of an event class. Use an appropriate and specific event class for its type.

```
public delegate void MouseEventHandler(Object^ sender, MouseEvent^ e);
```

- Do name event argument classes with the `EventArgs` suffix.

```
public ref class MouseEventArgs : EventArgs {
   int x;
   int y;

public:
   MouseEventArgs(int x, int y) {
      this->x = x;
      this->y = y;
   }

   property int X { int get() { return x; } }
   property int Y { int get() { return y; } }
};
```

- Do name event names that have a concept of pre- and post-operation using the present and past tense (do not use `BeforeXxx`/`AfterXxx` pattern). For example, a close event that could be canceled would have a `Closing` and `Closed` event.

```
event ControlEventHandler^ ControlAdded;
```

- Consider naming events with a verb.

## D.12 Case sensitivity

- Don't use names that require case sensitivity. Components might need to be usable from both case-sensitive and case-insensitive languages. Since case-insensitive languages cannot distinguish between two names within the same context that differ only by case, components must avoid this situation.

Examples of what not to do:

- Don't have two namespaces whose names differ only by case.

```
namespace ee::cummings;
namespace Ee::Cummings;
```

- Don't have a function with two parameters whose names differ only by case.

```
void F(String^ a, String^ A)
```

- Don't have a namespace with two types whose names differ only by case.

```
System::WinForms::Point p;
System::WinForms::POINT pp;
```

- Don't have a type with two properties whose names differ only by case.

```
property int f { int get(); void set(int value); }
property int F { int get(); void set(int value); }
```

- Don't have a type with two functions whose names differ only by case.

```
void f();
void F();
```

## D.13 Avoiding type name confusion

Different languages use different names to identify the fundamental CLI types, so in a multi-language environment, designers must take care to avoid language-specific terminology. This subclause describes a set of rules that help avoid type name confusion.

- Do use semantically interesting names rather than type names.
- In the rare case that a parameter has no semantic meaning beyond its type, use a generic name. For example, a class that supports writing a variety of data types into a stream might have:

```
void Write(double value);
void Write(float value);
void Write(long long value);
void Write(int value);
void Write(short value);
```

rather than a language-specific alternative such as:

```
void Write(double doubleValue);
void Write(float floatValue);
void Write(long long longlongValue);
void Write(int intValue);
void Write(short shortValue);
```

- In the extremely rare case that it is necessary to have a uniquely named function for each fundamental data type, do use the following universal type names: `SByte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Boolean`, `Char`, `String`, and `Object`. For example, a class that supports reading a variety of data types from a stream might have:

```
double ReadDouble();
float ReadSingle();
long long ReadInt64();
int ReadInt32();
short ReadInt16();
```

rather than a language-specific alternative such as:

```
double ReadDouble();
float ReadFloat();
long long ReadLongLong();
int ReadInt();
short ReadShort();
```

**End of informative text**

# Annex E. Future directions

**This annex is informative.**

This annex contains information about features that might be considered for a future revision of this Standard.

## E.1 Static members in interfaces

Yet to come.[[#176]]

## E.2 Mixed types

Yet to come. [[#176]]

## E.3 gcnew of unmanaged types

Yet to come. [[#176]]

## E.4 new of managed types

Yet to come. [[#176]]

## E.5 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.

## E.6 Literals

Investigate whether string literals can include compile-time expressions, such as concatenation of strings with non-strings using the + operator.

## E.7 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 helps 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 must 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* has been extended 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*]

Add text to show the behavior in the CLI (including CIL).

## E.8 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.

**End of informative text**

# Annex F. Incompatibilities with Standard C++

**This annex is informative.**

This annex contains information about aspects of C++/CLI that are incompatible with Standard C++.

1. Commas in [], but not having enclosing parentheses, being treated as punctuators rather than as operators.[[Ed.]]
2. New keywords: gcnew, nullptr. [[Ed.]]
3. Exception handling stuff.[[#178]]
4. Treatment of >> and >>=.[[Ed.]]

**End of informative text**

# Annex G. Index

**This annex is informative.**