A Design Rationale for C++/CLI

Version 1.1 — February 24, 2006
(later updated with minor editorial fixes)

Herb Sutter (hsutter@microsoft.com)

---

1 Overview

> *A multiplicity of libraries, run-time environments, and development environments are essential to support the range of C++ applications. This view guided the design of C++ as early as 1987; in fact, it is older yet. Its roots are in the view of C++ as a general-purpose language*
>
> *— B. Stroustrup (D&E, p. 168)*

C++/CLI was created to enable C++ use on a major run-time environment, ISO CLI (the standardized subset of .NET).

A technology like C++/CLI is essential to C++'s continued success on Windows in particular. CLI libraries are the basis for many of the new technologies on the Windows platform, including the WinFX class library shipping with Windows Vista which offers over 10,000 CLI classes for everything from web service programming (Communication Foundation, WCF) to the new 3D graphics subsystem (Presentation Foundation, WPF). Languages that do not support CLI programming have no direct access to such libraries, and programmers who want to use those features are forced to use one of the 20 or so other languages that do support CLI development. Languages that support CLI include COBOL, C#, Eiffel, Java, Mercury, Perl, Python, and others; at least two of these have standardized language-level bindings.

C++/CLI's mission is to provide direct access for C++ programmers to use existing CLI libraries and create new ones, with little or no performance overhead, with the minimum amount of extra notation, and with full ISO C++ compatibility.

1.1 Key Goals

Enable C++ to be a first-class language for CLI programming.

Support important CLI features, at minimum those required for a *CLS consumer* and *CLS extender*: CLI defines a Common Language Specification (CLS) that specifies the subsets of CLI that a language is expected to support to be minimally functional for consuming and/or authoring CLI libraries.

Enable C++ to be a systems programming language on CLI: A key existing strength of C++ is as a systems programming language, so extend this to CLI by leaving no room for a CLI language lower than C++ (besides ILASM).

Use the fewest possible extensions.

Require zero use of extensions to compile ISO C++ code to run on CLI: C++/CLI requires compilers to make ISO C++ code "just work" — no source code changes or extensions are needed to compile C++ code to execute on CLI, or to make calls between code compiled "normally" and code compiled to CLI instructions.

Require few or no extensions to consume existing CLI types: To use existing CLI types, a C++ programmer can ignore nearly all C++/CLI features, and typically writes a sprinkling of **gcnew** and ^ (see also Appendix, page 54). Most C++/CLI extensions are used only when authoring new CLI types.

Use pure conforming extensions that do not change the meaning of existing ISO C++ programs and do not conflict with ISO C++ or with C++0x evolution: This was achieved nearly perfectly, including for macros.

Be as orthogonal as possible.

Observe the principle of least surprise: If feature X works on C++ types it should also seamlessly work on CLI types, and vice versa. This was mostly achieved, notably in the case of templates, destructors, and

other C++ features that do work seamlessly on CLI types; for example, a CLI type can be templated and/or be used to instantiate a template, and a CLI generic can match a template template parameter.

Some unifications were left for the future; for example, a contemplated extension that the C++/CLI design deliberately leaves room for is to use **new** and * to (semantically) allocate CLI types on the C++ heap, making them directly usable with existing C++ template libraries, and to use **gcnew** and ^ to (semantically) allocate C++ types on the CLI heap. (See §3.3.3.) Note that this would be highly problematic if C++/CLI had not used a separate **gcnew** operator and ^ declarator to keep CLI features out of the way of ISO C++.

1.2 Basic Design Forces

Four main programming model design forces are mentioned repeatedly in this paper:

1. It is necessary to add language support for a key feature that semantically cannot be expressed using the rest of the language and/or must be known to the compiler.

> *Classes can represent almost all the concepts we need. … Only if the library route is genuinely infeasible should the language extension route be followed.*
>
> *— B. Stroustrup (D&E, p. 181)*

In particular, a feature that unavoidably requires special code generation must be known to the compiler, and nearly all CLI features require special code generation. Many CLI features also require semantics that cannot be expressed in C++. Libraries are unquestionably preferable wherever possible, but either of these requirements rules out a library solution.

Note that language support remains necessary even if the language designer smoothly tries to slide in a language feature dressed in library's clothing (i.e., by choosing a deceptively library-like syntax). For example, instead of

```
property int x;        // A: C++/CLI syntax
```

the C++/CLI design could instead have used (among many other alternatives) a syntax like

```
property<int> x;       // B: an alternative library-like syntax
```

and some people might have been mollified, either because they looked no further and thought that it really was a library, or because they knew it wasn't a library but were satisfied that it at least looked like one. But this difference is entirely superficial, and nothing has really changed — it's still a language feature and a language extension to C++, only now a deceitful one masquerading as a library (which is somewhere between a fib and a bald-faced lie, depending on your general sympathy for magical libraries and/or grammar extensions that look like libraries).

In general, even if a feature is given library-like syntax, it is still not a true library feature when:

- the name is recognized by the compiler and given special meaning (e.g., it's in the language grammar, or it's a specially recognized type); and/or
- the implementation is "magical."

Either of these make it something no user-defined library type could be. Note that, in the case of surfacing CLI properties in the language, at least one of these must be true even if properties had been exposed using syntax like B. (For more details and further discussion of alternatives, see §3.2.)

Therefore choosing a syntax like B would not change anything about the technical fact of language extension, but only the political perception. This approach amounts to dressing up a language feature with library-like syntax that pretends it's something that it can't be. C++'s tradition is to avoid magic libraries

and has the goal that the C++ standard library should be implementable in C++ without compiler collusion, although it allows for some functions to be intrinsics known to the compiler or processor. C++/CLI prefers to follow C++'s tradition, and it uses magical types or functions only in four isolated cases: **cli::array**, **cli::interior_ptr**, **cli::pin_ptr**, and **cli::safe_cast**. These four can be viewed as intrinsics — their implementations are provided by the CLI runtime environment and the names are recognized by the compiler as tags for those CLI runtime facilities.

2. It is important not only to hide unnecessary differences, but also to expose essential differences.

I try to make significant operations highly visible. — B. Stroustrup (D&E, p. 119)

First, an unnecessary distinction is one where the language adds a feature or different syntax to make something look or be spelled differently, when the difference is not material and could have been "papered over" in the language while still preserving correct semantics and performance.

For example, CLI reference types can never be physically allocated on the stack, but C++ stack semantics are very powerful and there is no reason not to allow the lifetime semantics of allocating an instance of a reference type **R** on the stack and leveraging C++'s automatic destructor call semantics. C++/CLI can, and therefore should, safely paper over this difference and allow stack-based semantics for reference type objects, thus avoiding exposing an unnecessary distinction. Consider this code for a reference type **R**:

```
void f() {
  R r;                  // ok, conceptually allocates the R on the stack
  r.SomeFunc();         // ok, use value semantics

  ...

}                       // destroy r here
```

In the programming model, **r** is on the stack and has normal C++ stack-based semantics. Physically, the compiler emits something like the following: (See §3.1.3 for more information.)

```
// f, as generated by the compiler
void f() {
  R^ r = gcnew R;   // actually allocated on the CLI heap
  r->SomeFunc();    // actually uses indirection

  ...
  delete r;         // destroy r here (memory is reclaimed later)
}
```

Second, it is equally important to avoid obscuring essential differences, specifically not try to "paper over" a difference that actually matters but where the language fails to add a feature or distinct syntax.

For example, although CLI object references are similar to pointers (e.g., they are an indirection to an object), they are nevertheless semantically not the same because they do not support all the operations that pointers support (e.g., they do not support pointer arithmetic, stable values, or reliable comparison). Pretending that they are the same abstraction, when they are not and cannot be, causes much grief. One of the main flaws in the Managed Extensions design is that it tried to reduce the number of extensions to C++ by reusing the * declarator, where **T*** would implicitly mean different things depending the type of **T** — but three different and semantically incompatible things, lurking together under a single syntax. (See §3.3.2.)

The road to unsound language design is paved with good intentions, among them the papering over of essential differences.

3. Some extensions actively help avoid getting in the way of ISO C++ and C++0x evolution.

Any compatibility requirements imply some ugliness. — B. Stroustrup (D&E, p. 198)

A real and important benefit of extensions is that using an extension that the ISO C++ standards committee (WG21) has stated it does not like and is not interested in can be the best way to stay out of the way of C++0x evolution, and in several cases this was done explicitly at WG21's direction.

For example, consider the extended **for** loop syntax: C++/CLI stayed with the syntax **for each(T t in c)** after consulting the WG21 evolution working group at the Sydney meeting in March 2004 and other meetings, where EWG gave the feedback that they were interested in such a feature but they disliked both the **for each** and **in** syntax and were highly likely never to use it, and so directed C++/CLI to use the undesirable syntax in order to stay out of C++0x's way. (The liaisons noted that if in the future WG21 ever adopts a similar feature, then C++/CLI would want to drop its syntax in favor of the WG21-adopted syntax; in general, C++/CLI aims to track C++0x.)

Using an extension that WG21 might be interested in, or not using an extension at all but adding to the semantics of an existing C++ construct, is liable to interfere with C++0x evolution by accidentally constraining it. For another example, consider C++/CLI's decision to add the **gcnew** operator and the **^** declarator. This paper later discusses the technical rationale for this feature in more depth, but for now consider just the compatibility issue: By adding an operator and a declarator that are highly likely never to be used by ISO C++, C++/CLI avoids conflict with future C++ evolution (besides making it clear that these operations have nothing to do with the normal C++ heap). If C++/CLI had instead specified a **new (gc)** or **new (cli)** "placement new" as its syntax for allocation on the CLI heap, that choice could have conflicted with C++0x evolution which might want to provide additional forms of placement **new**. And, of course, using a placement syntax could and would also conflict with existing code that might already use these forms of placement **new** — in particular, **new (gc)** is already used with the popular Boehm collector. (For other reasons why such a syntax causes technical problems, see §3.3.)

4. Users rely heavily on keywords, but that doesn't mean the keywords have to be reserved words.

My experience is that people are addicted to keywords for introducing concepts to the point where a concept that doesn't have its own keyword is surprisingly hard to teach. This effect is more important and deep-rooted than people's vocally expressed dislike for new keywords. Given a choice and time to consider, people invariably choose the new keyword over a clever workaround.

— B. Stroustrup (D&E, p. 119)

When a language feature is necessary, programmers strongly prefer keywords. Normally, all C++ keywords are also reserved words, and taking a new one would break code that is already using that word as an identifier (e.g., as a type or variable name).

C++/CLI avoids adding reserved words so as to preserve the goal of having pure extensions, but it also recognizes that programmers expect keywords. C++/CLI balances these requirements by adding keywords where most are not reserved words and so do not conflict with user identifiers (see Figure 1 and Figure 2, taken from [N1557] slides 7, 8, and 56):[1]

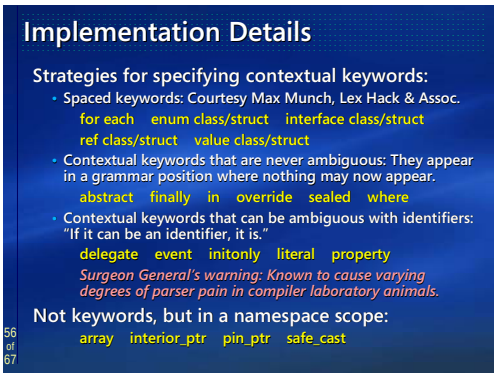


Figure 1: Contextual keywords (from [N1557])

[1] For related discussion see also my blog article "C++/CLI Keywords: Under the hood" (November 23, 2003).

- **Spaced keywords.** These are reserved words, but cannot conflict with any identifiers or macros that a user may write because they include embedded whitespace (e.g., **ref class**).
- **Contextual keywords.** These are special identifiers instead of reserved words. Three techniques were used: 1. Some do not conflict with identifiers at all because they are placed at a position in the grammar where no identifier can appear (e.g., **sealed**). 2. Others can appear in the same grammar position as a user identifier, but conflict is avoided by using a different grammar production or a semantic disambiguation rule that favors the ISO C++ meaning (e.g., **property**, **generic**), which can be informally described as the rule "if it can be a normal identifier, it is." 3. Four "library-like" identifiers are considered keywords when name lookup finds the special marker types in namespace **cli** (e.g., **pin_ptr**).

Note these make life harder for compiler writers, but that was strongly preferred in order to achieve the dual goals of retaining near-perfect ISO C++ compatibility by sticking to pure extensions and also being responsive to the widespread programmer complaints about underscores (see §1.3).

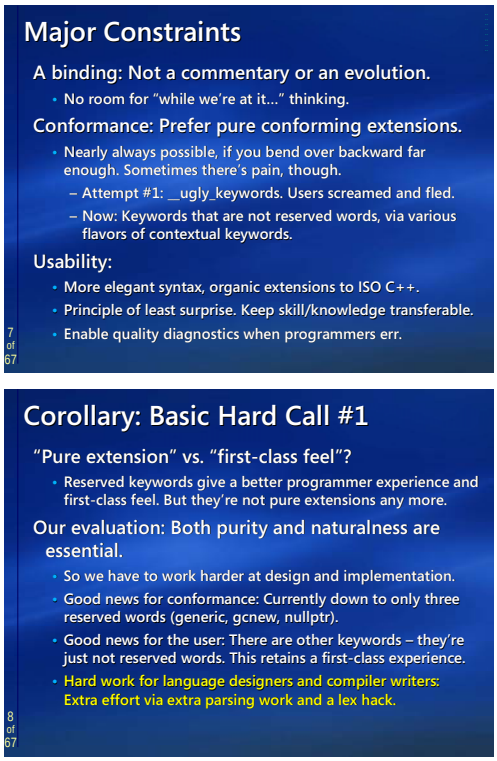


Figure 2: Avoiding both incompatibility and underscores via contextual keywords in C++/CLI (from [N1557]; note that later generic also became a contextual keyword rather than a reserved word)

1.3 Previous Effort: Managed Extensions

C++/CLI is the second publicly available design to support CLI programming in C++. The first attempt was Microsoft's proprietary Managed Extensions to C++ (informally known as "Managed C++"), which was shipped in two releases of Visual C++ (2002 and 2003) and continues to be supported in deprecated mode in Visual C++ 2005.

Because the Managed Extensions design deliberately placed a high priority on C++ compatibility, it did two things that were well-intentioned but that programmers objected to:

- The Managed Extensions wanted to introduce as few language extensions as possible, and ended up reusing too much existing but inappropriate C++ notation (e.g., ***** for pointers). This caused serious problems where it obscured essential differences, and the design for overloaded syntaxes like ***** was both technically unsound and confusing to use.
- The Managed Extensions scrupulously used names that the C++ standard reserves for C++ implementations, notably keywords that begin with a double underscore (e.g., **__gc**). This caused unexpectedly strong complaints from programmers, who made it clear that they hated writing double underscores for language features.

Many C++ programmers tried hard to use these features, and most failed. Having the Managed Extensions turned out to be not significantly better for C++ than having no CLI support at all. However, the Managed Extensions did generate much direct real-world user experience with a shipping product about what kinds of CLI support did and didn't work, and why; and this experience directly informed C++/CLI.

1.4 Organization of This Paper

This paper captures a small but representative sample of the experience gained by a number of C++ experts who have worked on defining bindings between C++ and CLI.

Section 2 gives a brief overview of issues involved with compiling for a CLI target.

Section 3 then considers several specific features, chosen as representative examples that cover most CLI feature areas. The discussion of each feature includes the relevant CLI requirements, the Managed Extensions design and the experience gained with that in the field, the C++/CLI design and rationale with notes about sample design alternatives, and consideration of how exposing the feature via keywords, reserved words, or library extensions is appropriate or inappropriate:

- **CLI types (e.g., ref class, value class):** Why new type categories are needed, and considerations for choosing the right defaults for CLI types.
- **CLI type features (e.g., property):** Why new abstractions are needed for some CLI features.
- **CLI heap (e.g., ^, gcnew):** Why to add a new declarator and keyword.
- **CLI generics (generic):** Why the new genericity feature is distinct from templates, but compatible and highly integrated with templates.
- **C++ features (e.g., template, const):** Why and how these are made to work on CLI types.

Finally, section 4 considers some frequently asked questions about C++/CLI.

2 Compiling for CLI: A Brief Survey

CLI programs are represented as:

- **Code instructions:** Plain code execution control just like you'd find in any typical CPU instruction set, including compare, branch, call a subroutine, and so on.
- **Metadata:** User-defined types, including inheritance, operators, generics, and so on.

2.1 Compiling an ISO C++ Program to Metadata

An ISO C++ program doesn't have any CLI types in it, so compiling any ISO C++ program to target CLI basically involves just emitting CLI instructions. An ISO C++ program can be compiled as-is to CLI instructions, including full use of all C++ features including the standard library, multiple inheritance, templates, optional garbage collection for the C++ heap, and other features. For example, consider "hello world":

```
#include <iostream>
int main() {
  std::cout << "Hello, world!" << std::endl;
}
```

Compiling this code for x86 or for CLI yields similar disassembly listings:

```
// x86 disassembly
_main           PROC
    push        ebp
    mov         ebp, esp
    push        OFFSET
?endl@std@@YAAAV?$basic_ostream@DU?$char
_traits@D@std@@@1@AAV21@@Z ; std::endl
    push        OFFSET $SG13670
    push        OFFSET
?cout@std@@3V?$basic_ostream@DU?$char_tra
its@D@std@@@1@A ; std::cout
    call
    ??$?6U?$char_traits@D@std@@@std@@YAA
AV?$basic_ostream@DU?$char_traits@D@std@
@@0@AAV10@PBD@Z ;
std::operator<<<std::char_traits<char> >
    add         esp, 8
    mov         ecx, eax
    call
    ??6?$basic_ostream@DU?$char_traits@D@st
d@@@std@@QAEAAV01@P6AAAV01@AAV01@
@Z@Z ; std::basic_ostream<char,std::char_traits
<char> >::operator<<
    xor         eax, eax
    pop         ebp
    ret         0
_main           ENDP
```

```
// CLI disassembly (ILASM code)
.method assembly static int32 main() IL managed {
  .vtentry 34 : 1
  .maxstack  2
  IL_0000:  ldsflda    valuetype
std.'basic_ostream<char,std::char_traits<char> >'* __imp_std.cout
  IL_0005:  ldind.i4
  IL_0006:  ldsflda    valuetype
'<CppImplementationDetails>'.$ArrayType$$$BY0M@$$CBD
'?A0xd1f5badc.unnamed-global-1'
  IL_000b:  call       valuetype
std.'basic_ostream<char,std::char_traits<char> >'*
'std.operator<<<struct std::char_traits<char> >'(valuetype
std.'basic_ostream<char,std::char_traits<char> >'*, int8 *)
  IL_0010:  ldsfld     int32**
__unep@?endl@std@@$$FYAAAV?$basic_ostream@DU?$char_tra
its@D@std@@@1@AAV21@@Z$PST04000053
  IL_0015:  call       valuetype
std.'basic_ostream<char,std::char_traits<char> >'* 'std.basic_-
ostream<char,std::char_traits<char> >.<<'(valuetype std.'basic-
_ostream<char,std::char_traits<char> >'*, method unmanaged
cdecl valuetype std.'basic_ostream<char,std::char_traits<char> >'*
*(valuetype std.'basic_ostream<char,std::char_traits<char> >'*))
  IL_001a:  pop
  IL_001b:  ldc.i4.0
  IL_001c:  ret
}
```

So, for plain C++ code, targeting the CLI virtual machine can be handled like targeting a processor instruction set. (CLI later compiles its instructions in turn to the instruction set of the actual CPU present at run time.)

Note that as an implementation detail all C++ types are typically emitted as opaque array-of-bytes CLI value types whose internal members are not visible to the CLI environment. For example, this code:

```
class C {
  int i;

  void f() { ... };
  void g() { ... };
  // ...
};
```

is encoded in metadata as an opaque value type with an explicit size and layout (as opposed to the usual CLI default of having the JIT compiler determine the object layout depending on the user's execution environment) and whose members are all hidden from the CLI runtime, that looks something like this:

```
// A C++ class is emitted as an opaque value type (note the absence of f and g)
.class private sequential ansi sealed beforefieldinit C
  extends [mscorlib]System.ValueType
{
  .pack 0
  .size 4
  .custom instance void ... MiscellaneousBitsAttribute::.ctor(int32) = ( 01 00 40 00 00 00 00 00 )
  .custom instance void ... Runtime.CompilerServices.NativeCppClassAttribute::.ctor() = ( 01 00 00 00 )
  .custom instance void ... DebugInfoInPDBAttribute::.ctor() = ( 01 00 00 00 )
}
```

2.2 Compiling C++/CLI Extensions to Metadata

For a C++/CLI program that authors CLI types, the compiler additionally has to emit the correct metadata that describes those types. For example, consider this simple CLI reference type:

```
ref class R {
public:
  property int x;
};
```

The C++/CLI compiler has to generate metadata code like this:

```
// A CLI type or feature is emitted using the corresponding metadata
.class private auto ansi beforefieldinit R
       extends [mscorlib]System.Object
{
  .field private int32 '<backing_store>x'
  .method public hidebysig specialname rtspecialname instance void .ctor() IL managed { ... }
  .method public hidebysig specialname instance int32 get_x() IL managed { ... }
  .method public hidebysig specialname instance void set_x(int32 __set_formal) IL managed { ... }
  .property instance int32 x() {
    .get instance int32 R::get_x()
    .set instance void R::set_x(int32)
  }
}
```

This paper will show that generating metadata like **.property** blocks often requires knowledge that the source code is using a CLI type or feature rather than just C++ facilities, and that some CLI abstractions

have to be reflected in the language just to be able to express them correctly in the generated metadata. Further, where CLI features have different semantics from the C++ features, they also cannot be represented correctly in standard C++ alone. (Before looking at the rationale for **property** in §3.2, consider whether it is possible to write a C++ library that would result in the above metadata to be generated, and what help you would need from the compiler to write such a library.)

For completeness, note that a CLI type can be made visible to a C++/CLI program in one of two ways:

- As C++ source code, typically via a **#include** of the class definition.
- As metadata, via a **#using** of the assembly the type was compiled to (treating it like any CLI type the program wants to use).

One issue that had to be considered throughout was that these must end up being the same — namely, that the same C++ type made accessible using either inclusion model had to have the same meaning. As an extra twist, many CLI tools and compilers don't read private metadata, but private metadata is important for C++ (e.g., it's important to know the names and signatures of private members when there are friends, and to apply C++ name lookup rules).

Finally, note that where the above issues affect C++/CLI-specific features that are represented in metadata but that are not part of CLS, they also affect portability between C++/CLI implementations. Of course CLI is specifically designed to be a *Common* Language Infrastructure where even completely different languages can interoperate, so two C++/CLI implementations would more or less automatically be compatible in the CLS subset. But it would be a shame if it was not guaranteed that CLI libraries produced by two different C++/CLI compilers will be fully interoperable also in their implementations of C++/CLI-specific extensions. It is important for C++/CLI to set a standard way for compilers to store and read metadata for C++/CLI-specific extensions like **const** that are stored as modopts, modreqs, or other special attributes, so that different implementations of C++/CLI can recognize and preserve their meanings. (Note: C++/CLI does not specify binary compatibility between the pure C++ parts of the two implementations, which is left unspecified by the C++ standard.)

3 Design Examples In Depth

This section considers specific CLI features and presents:

- The CLI feature and basic requirements, including where applicable representative metadata that must be generated.
- The Managed Extensions design shipped since Visual C++ 2002, which used fewer extensions, and in what ways it proved to be insufficient.
- The C++/CLI design and rationale.
- Other major design choices that were considered, both in earlier iterations of C++/CLI, and in other design efforts that were abandoned.

3.1 CLI Types (e.g., ref class, value class)

C++/CLI adds the **ref class** and **value class** type categories because CLI types do not behave exactly the same way as C++ types. (For a summary of some issues expanded upon in this section, see Figure 3, taken from [N1557] slide 10.)

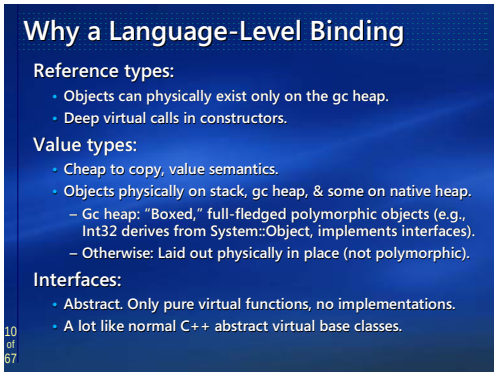


Figure 3: Unique features of the CLI types (from [N1557])

CLI classes and C++ classes share most features in common with obvious and familiar meaning. For example, C++ programmers will not be surprised that CLI types support inheritance, member variables, virtual functions, different accessibility on different members, user-defined operators like **operator+**, and so on. Most of the differences that do exist are small enough that they can be seamlessly papered over in a way that preserves all correct semantics and so doesn't hide any essential differences.

This section discusses some differences between the C++ and CLI type systems and focuses primarily on the relationship among the **class**, **ref class**, and **value class** abstractions, with incidental mention of **interface class**. An addendum to this section covers choosing the right defaults for CLI types. This section does not discuss related topics like **enum class** or **interface class** in detail.

3.1.1 Basic Requirements

CLI has two major kinds of types:

- **CLI reference types.** These inherently have reference semantics rather than value semantics, and so unlike C++ types they are not copy-constructible by default, or at all without some work — CLI itself has no notion of copy construction, and instead the usual convention is for the class author to write an explicit **Clone** virtual function that outside code can call to create a new instance initialized with the original object's values. CLI reference type objects can only physically exist on the CLI heap.
- **CLI value types.** These have a dual nature. Value types are intended to represent simple values like numbers that do not have unique object identity and are bitwise-copyable. Normally value objects exist in their "unboxed" form where they are laid out directly on the stack or, when members of an object, directly embedded in the layout of an object (just like C++ members that are held by value); in this form, they are not real big-Oh **Object**s and do not have vtables. When they need to be treated like full-fledged objects (e.g., to perform a call to a CLI interface that the value type imple-

ments), they can be converted into a "boxed" form where they are represented as a real object of CLI reference type on the CLI heap, and in this form they do have a vtable. In the CLI, both the boxed and unboxed forms have the same name, and there is no way to directly distinguish between the two forms programmatically in CLI or in most CLI languages (but there is in C++/CLI; see §3.3.3). CLI value types inherit from **System::ValueType** and cannot be further derived from. Note that, as with CLI reference types, copy construction makes no sense for CLI value types either but for a different reason, namely the bitwise-copyable assumption; there are places in the CLI where the virtual machine itself can make bitwise copies, but where the compiler cannot insert function calls in order to faithfully simulate copy construction semantics consistently.

The following are rules that are common to both kinds of CLI types, but which differ from the rules of C++ types:

- **Virtual function dispatch during construction is deep.** When a constructor calls a virtual member function of its own object, the virtual dispatch is deep — dispatch is to the most-derived override available in the actual type of the object being constructed, even if that type's constructor has not yet been run. (C++ used to follow the same rule, but long ago switched to making this virtual dispatch shallow — in C++, dispatch is virtual but only to the most-derived override available in the base class currently being constructed. This makes the implementation of construction slightly costlier because it can require fixups to vtables, as each further-derived constructor body first resets the vtable to point to the overrides that should be visible at that level, but it is arguably safer because it avoids callings member functions of unconstructed objects.)
- **Inheritance is always single.** A CLI type always has exactly one base class, even if it is just **System::Object** (the root of the monolithic CLI type hierarchy). However, a CLI type can implement any number of CLI interfaces.
- **Inheritance is always public.** A CLI type's base class is always public base class. It is not possible for a CLI type to inherit nonpublicly from another CLI type. Indeed, the public nature of inheritance is so ingrained in CLI that "public" is not even represented in the metadata or in the ILASM grammar (the CLI instruction language assembler); "public" is just what must always be.

There are other small semantic differences between CLI types and C++ types, but those are some of the major ones. Neither of these kinds of CLI types can be surfaced correctly in the programming model as C++ classes, because they do not behave exactly the same way as C++ classes do, and this by itself makes new class categories essentially required. In addition, it turns out that the general type of each class also needs to be known to a C++ compiler in a class *forward declaration*, not just an actual class definition, and that by itself too is sufficient to make a new class category essentially required.[2]

This section will focus primarily on CLI reference and value types, but note also that CLI interfaces are a third important kind of CLI abstraction. Although they have some subtle semantics that have no exact analog in C++, it is almost correct to think of a CLI interface as being the same as a C++ abstract base class that is made up of public pure virtual functions and that can inherit from zero or more other such abstract base classes.

[2] Note that the potential future unification of allocating any type on either the C++ or CLI heap (see §3.3.3) requires knowledge of the type category in forward declarations, because that knowledge is needed to even declare a **T*** and **T^** which would have different implementations depending on the type of **T** (but compatible ones with consistent semantics this time, unlike the Managed Extensions).

3.1.2 Managed Extensions Design

The Managed Extensions exposed the CLI types as new class categories. In the Managed Extensions design:

- A CLI reference type is declared with **__gc class** (or **__gc struct**).
- A CLI value type is declared with **__value class** (or **__value struct**).
- A CLI interface is declared with **__gc __interface**. All member functions are implicitly public and pure virtual; the keyword **virtual** and the suffix **=0** are allowed but not required. A **__gc __interface** cannot have data members, static members, nonpublic members, or implementations of member functions.

The Managed Extensions followed C++'s **class**/**struct** tradition: Analogously to C++, the difference between a **__gc struct** and a **__gc class** (and **__value struct** and a **__value class**) is that the default accessibility of members and base classes is public for the "struct" form, but private for the "class" form.

For example:

```
// Managed Extensions syntax
__gc __interface I { };          // CLI interface
__gc class R : public I { };     // CLI reference type
__gc class R2 : R { };           // inheritance is public by default (writing "public" is legal but redundant)
__value class V { };             // CLI value type
```

The Managed Extensions did impose restrictions on CLI types. In particular (among others not mentioned here):

- A CLI type cannot have a user-defined destructor, but only a user-defined finalizer. (Unfortunately, the finalizer was wrongly called a "destructor" and used the destructor's **~T** syntax in the Managed Extensions specification, in both cases following the mistakes made by other languages; this is fixed in C++/CLI.)
- A CLI type cannot be a template, or be used to instantiate a template.
- A CLI type cannot have a user-defined copy constructor.
- A CLI type cannot inherit from a C++ class or vice versa.
- A **__gc class** cannot define its own **operator new**, because it is always allocated by the CLI's own runtime allocator.

There were other reasons the Managed Extensions had to expose new categories of classes, even besides the requirement of representing behavioral differences. For example, consider that in the Managed Extensions a **T*** is actually implemented as either a normal C++ pointer, a CLI object reference, or a CLI interior pointer, depending on whether **T** is a C++ class, CLI reference type, or CLI value type (see §3.3.2). Then consider what the compiler should do with this code:

```
// alternative: what if CLI types were declared with just "class" (note: not Managed Extensions syntax)
class C;                         // forward declaration — class category of C is so far unknown
int main() {
  C* p;                          // what would the compiler do with this?
  ...
}
```

If all types were declared with just **class**, including in forward declarations, then there would be insufficient information for the compiler to know what to do here, because the compiler has to know what

kind of pointer implementation to generate for **p**. (See also §3.1.4 for further discussion on options for forward declarations.) So *something* has to appear in the class declaration, and the minimum possible extension is to require the programmer to write one word (e.g., **__gc** or **__value**) once on the declaration of the class only, to tag that "that set of rules applies to this type," after which the code that uses the type just instantiates and uses objects as usual without any further syntax extensions. One word per type for the above-cited rules is about as low-impact as it's possible to get, and C++/CLI ended up following the same pattern.

A small but meaningful criticism of this syntax is that using a term like "**__gc**" to distinguish CLI reference types and interfaces emphasizes the wrong thing: The essential nature of these types is their reference semantics, not where they are allocated. (Besides, languages can choose to expose non-garbage-collected semantics for instances of reference types; for example, see §3.3.3 for details on how C++/CLI papers over a nonessential difference and allows stack-based lifetime semantics for instances of CLI reference types.)

Unlike many other CLI languages that hide pointers by making them implicit, a strength of C++ is that it makes indirection explicit (e.g., with *****) and so enables the programmer to distinguish between a pointer and a pointee. This is directly valuable for expressing the difference between the unboxed and boxed forms of a value type, including making boxing and unboxing operations visible:

```
// Managed Extensions syntax
__value class V { };
int main() {
  V v;                  // unboxed instance (V)
  V* p = new V;         // create a boxed instance; note explicit syntax for a boxed value type (V*)
  V v2 = *p;            // explicit unboxing
  *p = ...;             // explicit support for modifying a boxed value type in place (without copying)
}
```

This allows direct language support in C++ for strongly typed boxed values, including for modifying a boxed value type in place without copying, which is a feature not available in most other CLI languages. (See further notes in §3.1.3.)

3.1.3 C++/CLI Design and Rationale

3.1.3.1 Paper over difference that can be hidden ("it just works")

C++/CLI has a much smoother integration between CLI types and C++ features:

- A CLI type can have a real destructor, which as usual is spelled **~T** and is seamlessly mapped to the CLI **Dispose** idiom (note: this is nontrivial, and required C++/CLI to influence CLI to modify and regularize its pattern). This is perhaps the single most important strength of C++/CLI because it makes resource-owning CLI types far easier to use in C++ than in other CLI languages.[3] (CLI types can of course also have finalizers, and these are correctly distinguished from destructors and spelled as **!T**.) See Figure 4, taken from [N1557] slides 33-36.
- A CLI type can be a template, can have member function templates, and can be used to instantiate a template. (See §3.5.3.)

[3] For further discussion, see my blog articles "Destructors vs. GC? Destructors + GC!" (November 23, 2004) and "C++ RAII compared with Java Dispose pattern" (July 31, 2004).

- A **ref class** can have a user-defined copy constructor and/or copy assignment operator. These are emitted with modreqs (as are pass-by-value function parameters that use the copying semantics), and so CLI languages other than C++ that do not support such value semantics will ignore these special functions; therefore types authored this way will be usable in other languages, but those languages just won't allow access to the value copying behavior (which is usual in such languages).

To avoid surprise, C++/CLI likewise follows C++'s **class**/**struct** tradition where the only potential difference is the default accessibility of bases and members.

3.1.3.2 Expose essential differences

The C++/CLI design probably follows the Managed Extensions design more closely in this area than in any other. It likewise adds a single word on the class declaration, but it chooses cleaner names — ones that better convey the essential nature of CLI reference types. In particular, choosing **ref** correctly conveys that the essential nature of a CLI reference type is its reference semantics (see §3.1.2). C++/CLI also gets away from the double-underscores that programmers complained about (see §1.3). For example:

```
interface class I { };     // CLI interface
ref class R : public I { };// CLI reference type
ref class R2 : R { };      // inheritance is public by default
                           // ("public R" is legal but redundant)
value class V { };         // CLI value type
```

I chose to follow the naming convention of "***adjective* class**" for its natural syntactic consistency and because it also extended cleanly to **enum class** (which correctly connotes that the scoping and strong typing associated with the concept of a "class" apply to CLI enums; however, CLI enums are not further discussed in this paper):

```
          class C;     // or "struct"
    ref class R;       // or "ref struct"
  value class V;       // or "value struct"
interface class I;     // or "interface struct"
   enum class E;       // or "enum struct"
```

Note that some argued for breaking this syntactic symmetry by changing **interface class** to **ref interface**. (See §3.1.4.)

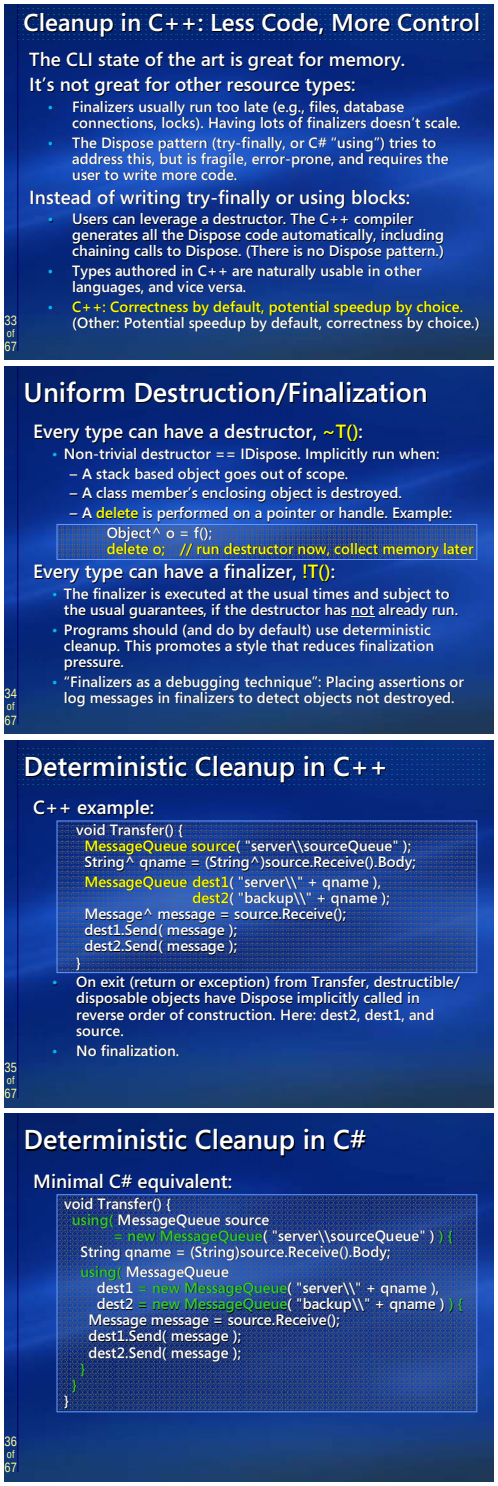


Figure 4: Notes on destructors for CLI types (from [N1557])

As with the Managed Extensions, this surfaces the CLI types with the minimum possible intrusiveness, where to denote a new CLI type the programmer writes one word (e.g., **ref** or **value**, once on the declaration of the class only) to tag that "that set of rules applies," after which the code that uses the type just

instantiates and uses objects as usual without any further syntax extensions besides ^ and **gcnew**. Note that this allows natural forward declarations (e.g., **ref class R;**), which is a requirement (see the discussion of forward declarations in §3.1.2).

Note, however, that C++/CLI does not support user-defined copy construction and copy assignment on value types. This is because it turns out that the assumption that CLI value type instances are bitwise copyable is inherent in CLI, and CLI has a few places where the runtime has latitude to make bitwise copies of value type instances but where the C++/CLI compiler cannot inject code to make a function call and so guarantee that the copy constructor or copy assignment operator will be called. Since the compiler can't guarantee that these functions will be called, it would be wrong to let programmers write them and have them only mostly work.

The other two features that were mentioned as not supported in the Managed Extensions are also not supported in current C++/CLI, but C++/CLI deliberately leaves room for them in the future:

- A CLI type cannot inherit from a C++ class or vice versa, but the door for this was deliberately left open as a potential and desirable future unification. (See "mixed types" in the next section.)
- A **__gc class** cannot define its own **operator new**, because it is always allocated by the CLI's own runtime allocator. Here two notable doors are left open for future unifications (see also §3.3.3):
  - In the future C++/CLI might want to complete the unification of allocating any type on any heap, and if a CLI type **R** can be allocated on the C++ heap then of course it would make perfect sense to allow the programmer to write an **R::operator new** that returns an **R*** and would be called (to allocate the proxy object on the C++ heap) whenever **new R** is used to semantically allocate an **R** on the C++ heap.
  - If experience shows that it can use useful for CLI types to be able to write their own custom allocation functions for allocation on the CLI heap, the obvious natural way to surface it would be to allow programmers to write **operator gcnew** which returns a ^.

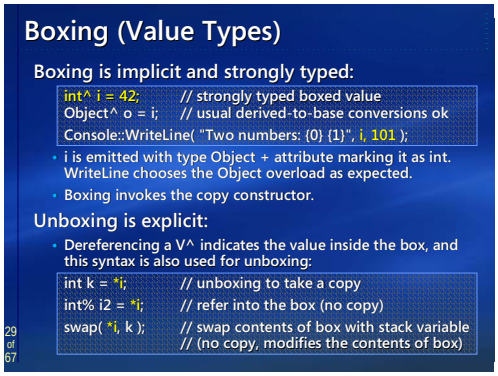


Figure 5: Direct and natural expression of boxed value types (from [N1557])

Finally, C++/CLI continues to take advantage of C++'s distinction between pointers and pointees and its explicit indirection. Like the Managed Extensions, C++/CLI supports a direct and natural expression of the difference between the unboxed and boxed forms of a value type, and the boxing and unboxing operations, this time via the ^ declarator and dereferencing:

```
value class V { };
int main() {
  V v;                  // unboxed instance (V)
  V^ p = gcnew V;       // boxed instance (V^)
  V v2 = *p;            // explicit unboxing
  *p = ...;             // explicit modify-in-place
}
```

This allows direct language support in C++ for strongly typed boxed values, including for modifying a boxed value type in place without copying, which is a feature not available in most other CLI languages. However, note that CLI has been evolving in the direction of value type instances being immutable, if only because many CLI languages and libraries have made this assumption and/or don't support a clear

distinction between the boxed and unboxed forms; for now, however, this direct and natural language expression of the dual nature of value types (**V** vs. **V^**) is a strength of C++ not available in other CLI languages and that shows C++'s usefulness as a system programming language on CLI. (See Figure 5, taken from [N1557] slide 29.)

3.1.3.3 Future unifications

In the future, it is possible to permit arbitrary cross-inheritance and cross-membership between C++ and CLI types (subject only to the CLI rule that a CLI type must have exactly one CLI base class).

Today, programmers who would like to have one type inherit from, or directly hold a member of, a second type from a different type category must hold the would-be base or member subobject as an indirectly held member and write the passthrough functions to "wire it up." For example, if we want to have a CLI type **R** inherit from a C++ class **C**, we would write something like:

```
ref class R {
  C* cImpl;              // indirectly hold what would
                         // be the C base class subobject
public:
  R() : cImpl( new C ) { }
  ~R() { !R(); }
  !R() { delete cImpl; cImpl = 0; }
  // ... etc. for other special functions ...
  int Foo( params ) { return cImpl->Foo( params ); }
  int Bar( params ) { return cImpl->Bar( params ); }
  // ... etc. for other functions we need passthroughs for ...
};
```

In the future it would be nice to just write:

```
// possible future unification
// (note: not currently C++/CLI)
ref class R : public C { };       // a mixed type
```

For example, in conjunction with the other potential extension of a unified heap (see §3.3.3), this would allow much useful flexibility, including:

- CLI types that inherit from C++ base classes and can be seamlessly passed to existing C++ code that uses pointers to those base classes.
- C++ types that inherit from CLI base classes or implement CLI interfaces (including generic base classes and interfaces) and can be seamlessly and safely passed to existing CLI code written in any CLI language.

The compiler can seamlessly represent this under the covers as a two-part object made up of a normal CLI object containing all of the CLI parts (e.g., the CLI base class, any CLI interfaces, any CLI members), and one C++ object containing the C++ parts (e.g., the C++ base classes and members).

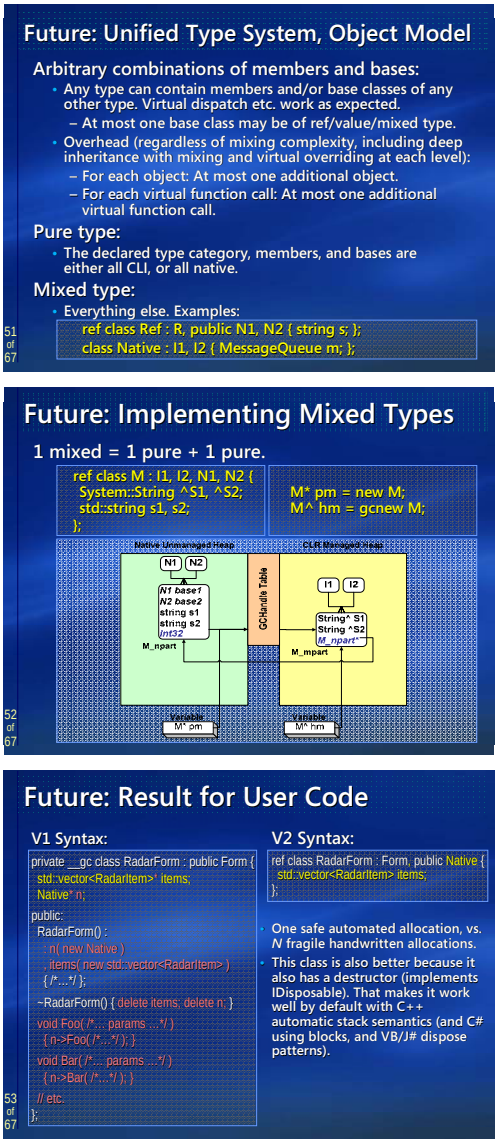


Figure 6: Mixed types as a possible future unification (from [N1557])

Note that there is much detail here that I'm eliding, such as the mechanics for a virtual member function of one kind of derived type to override a virtual member function of a different kind of base type, and how to then allow further-derived overriding by either kind of derived class without additional performance penalty. This can be done correctly and with good performance. (See Figure 6, taken from [N1557] slides 51-53.)

This strategy actually followed from a general implementation principle I adopted that greatly clarified our reasoning about how to expose the type system in the programming model. The implementation principle for heap-based objects was: "CLI objects are always physically on the CLI heap, C++ objects are always physically on the C++ heap," where we could then create programming model semantics on top of that implementation choice.[4] With this principle I deliberately closed the doors to unworkable flexibility that until then had been clung to, notably the latitude to lay out C++ objects physically on the CLI heap in the future (which can never work; see discussion in §3.3.3). This choice greatly clarified several aspects of the design that had been plagued by muddy thinking, and it greatly simplified the C++/CLI type system.

I remember the day I showed the idea for this potential unification on Bjarne Stroustrup's blackboard. I started to draw a picture like [N1557] slide 52, and without waiting for me to finish Bjarne went to his bookshelf and opened a book to show where, despite criticism, he had always insisted that C++ must not require objects to be laid out contiguously in a single memory block. Others had often questioned the usefulness of C++'s leaving that implementation door open for the future, but the mixed type design — where parts of the same logical object must be on different physical heaps — vindicates Bjarne's design judgment.

3.1.4 Other Alternatives (Sample)

3.1.4.1 Inheritance from System::Object or System::ValueType

One option that I tried hard to make work was to exploit the fact that CLI types have known base classes. In particular, all CLI types inherit from **System::Object**, and CLI value types inherit from **System::ValueType**. So "obviously" any type that inherits from those can be determined to be the corresponding kind of type.

This is extremely enticing, and one really wants to make it work (at least, I did). But it fails for several reasons, some of which will probably be obvious after having read the preceding sections.

First, although this option is fairly obvious for simple cases:

```
// inheritance-tagging rejected alternative (note: not C++/CLI)
class R : public System::Object { };          // aha, implicitly discover this to be a CLI reference type
class V : public System::ValueType { };       // aha, implicitly discover this to be a CLI value type
class C : public SomeNonCliType { };          // aha, implicitly discover this to be a C++ type
```

it quickly becomes too opaque and requires knowledge of base classes:

```
// inheritance-tagging rejected alternative (note: not C++/CLI)
class X : public Base { };                    // what kind of type is X? need to go look at Base...
```

So this alternative at least partly obscures an essential difference, namely that some behaviors are different for CLI types. To determine whether C++ or CLI behavior applies to a given class type, the programmer must search through its base classes. If that doesn't convince you, consider templates:

---

[4] Simple value types, including the fundamental types that are both C++ and CLI types (e.g., **int**), can exist in either heap.

```
// inheritance-tagging rejected alternative (note: not C++/CLI)
template<class Base>
class X : public Base { };      // X can be either a C++ or CLI type? — problematic
```

Here, the rules for virtual function lookup during construction (for example) would depend on what kind of type **Base** was. (Some people thought it might be cool to be able to write such a template that can be instantiated to be different kinds of types, but suffice it to say that this harbors subtle problems.)

That inheritance from some (possibly remote) ancestral base class should subtly affect behaviors like virtual function lookup in constructors seemed far too subtle in the end — it reminds me of Einstein's characterization of quantum entanglement as "spooky action at a distance," where here by analogy we don't want to expose the uncertainty of an inheritance entanglement to a distant ancestral base.

Additionally, this approach fails for other reasons, notably that the class category information must be known with just a class's forward declaration (see §3.1.1). That implies that base classes must become part of the forward declaration, which is just strange:

```
// forward declaration rejected alternative (note: not C++/CLI)
class X : public Base;          // ouch: require base classes to be listed on a forward declaration?
```

At this point one spends a few days vainly thinking about successively weirder options, such as using syntax like **class X : ref;** for forward declarations, using a tag to avoid listing the base classes:

```
// forward declaration rejected alternative (note: not C++/CLI)
class X : ref;                  // ouch: contextual keyword? could conflict with an actual base name?
```

There comes a point where you have to admit that you're trying too hard. The above was already essentially admitting it had to be an explicit class category, only under a needlessly strange syntax that also gratuitously introduced the potential for name clashes with actual base classes and was barely different from spelling it **ref class X;**). It is interesting, though, that starting down this "just see what it derives from" path independently leads one to something very like the syntax that was actually adopted.

Finally, at the time I considered this option I hadn't yet thought about the potential future unification of allowing mixed types having arbitrary membership and inheritance across the type system (see §3.1.3). When we got to mixed types I was glad that we hadn't pursued the alternative just described. It would unwittingly have closed the door on this future option for completely unifying the type system, because this alternative would have prevented allowing C++ types to inherit from CLI types. For example, consider that under the contemplated future unification of mixed types there is a difference between the following classes **R** and **C**, where **RBase** is a CLI type and **CBase** is a C++ type:

```
// possible future unification (note: not currently C++/CLI)
  ref class  R : public RBase, public CBase { };  // a CLI type that inherits from both RBase and CBase
      class  C : public RBase, public CBase { };  // a C++ type that inherits from both RBase and CBase
```

Here, if there was no separate class category for **ref class** and we tried to rely only on "inheritance from **System::Object**" as a tag to mark CLI types, only the second syntax would have been available, **C** would of necessity implicitly have been a CLI type, and there would be no way to express a type that is in every way a C++ type and follows C++ rules (e.g., for safe virtual calls) but happens to want to have a CLI base class (e.g., so as to be able to pass instances of this C++ type seamlessly to other CLI code).[5]

[5] For further discussion, see my blog entry "Why "ref class X", not just "class X : System::Object"?" (November 18, 2003).

3.1.4.2 ref interface

It's worth noting that some people argued at length in favor of spelling CLI interfaces as **ref interface** instead of **interface class**, on the grounds that:

- A CLI interface isn't really a class, even though it is similar to a C++ abstract base class.
- Both parts of the term **ref interface**, taken separately and together, arguably better conveys what a CLI interface is.

I ended up repeatedly rejecting this suggestion because I didn't agree that it was clearer, and it destroyed the syntactic symmetry and teachability of the "***adjective* class**" consistency without any proven advantage. I still think **interface class** is preferable and that it fits better with the rest of C++/CLI, but for completeness I should note this objection and point out that future experience could prove me wrong and that with "***adjective* class**" I could turn out to be clinging to a foolish consistency. I don't see such evidence yet as of this writing, however, and so far the consistency seems to be beneficial.

3.1.5 Defaults on C++ and CLI Types

*By definition, a **struct** is a class in which members are by default public; that is,*

struct s { ...

is simply shorthand for

class s { public: ...

*… Which style you use depends on circumstances and taste. I usually prefer to use **struct** for classes that have all data public.*

— B. Stroustrup (C++PL3e, p. 234)

The C++ **class** keyword is technically unnecessary, but was added for an essential reason: To change the default accessibility of members and bases. People sometimes complain that **class** and **struct** are the same thing, because the rationale is not obvious at first glance, but this choice is both correct and important — it is the only reasonable way to get both C compatibility and the right defaults for user-defined types.

In C, all **struct** members are public. To preserve compatibility with C code, C++ had to preserve that as the default for **struct**; there is no other choice. But public is a terrible default for a user-defined type, and a language that supports user-defined types ought instead to encourage strong encapsulation and data hiding by default. Having private as a default matters so much that Stroustrup correctly felt it was worth the high cost of taking a new keyword just to have a category of user-defined types where the default was private. Not only does that make it easier for programmers to do the right thing, but the default is important because it directly influences the mindset of programmers and the way they think about their code.

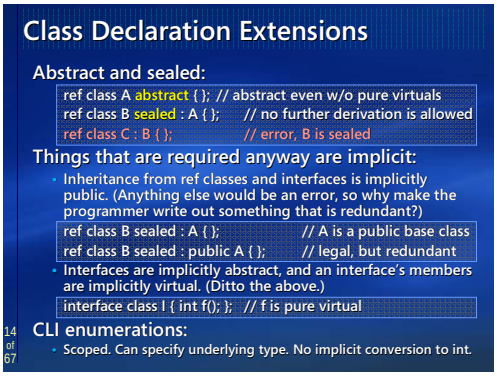


Figure 7: Why defaults matter (from [N1557], note middle of slide)

The result is that in C++ **struct** and **class** are identical abstractions with identical semantics differing only in the default accessibility of members and base classes.

In C++/CLI, a **ref class**'s members are private by default (the same as for a C++ **class**), but inheritance from another **ref class** or an **interface class** is implicitly public. Further, the member functions of an **interface class** are implicitly public and pure virtual. The reasons for having different defaults for CLI reference and interface types are stronger than the

reasons for having different accessibility defaults between C++'s **class** and **struct** (see also Figure 7, taken from [N1557] slide 14).

First, it is undesirable to allow defaults that are not only wrong but that can never be right. For example, a CLI reference type can only inherit from another CLI reference type publicly; there no possibility for non-public inheritance, to the point where **public** is not even written in the metadata or present in the ILASM assembler grammar, but is simply inherent and pervasively assumed.

If C++/CLI had required inheritance from base classes of a **ref class** to be private by default, then natural code like the following would be an error because the default accessibility would be an error, and the way for the programmer to make it compile would be to explicitly write **public** in the source code even though public inheritance is the only option there is or can ever be:

```
// rejected alternative to make the base class implicitly private (note: not C++/CLI)
ref class R2 : R {       // error, if the inheritance is implicitly private
};
```

Second, it is undesirable to force the programmer to explicitly specify something that admits no other option. For example, if **interface class** member functions were not implicitly virtual and abstract, the programmer would be gratuitously required to write **virtual** and **=0** (or **abstract**) on every member function declaration — and this would be wholly redundant, because they cannot be otherwise.

Setting inapplicable and invalid defaults on the basis of claimed consistency with the **class** defaults would be, in the words of Emerson, "a foolish consistency." Using the same defaults for CLI types, when those defaults can never be right and there is no real choice available to the programmer anyway, would be "consistent with C++" in only the most naïve sense — rather, it would be inconsistent with C++ and contrary to the spirit and sound design of C++, for C++ itself added **class** for the sole purpose of being able to set the right defaults.

3.2 CLI Type Features (e.g., property)

C++/CLI adds the **property** contextual keyword and abstraction because it is necessary to have compiler knowledge to recognize the abstraction and generate correct metadata.

This section discusses some differences between features that apply to CLI types and focuses primarily on **property**. This section does not mention related topics like indexed properties, **event**, and **override**.

3.2.1 Basic Requirements

A CLI property is a "virtual data member" that actually invokes get/set member functions. This lets languages give the illusion of modifying data members while actually safely going through functions, and allows tools to present richer interaction with objects.

For a C++/CLI program that authors CLI types, the compiler additionally has to emit the correct metadata that describes those types. For example, given a property like:

```
// C++/CLI syntax
property int x;
```

The C++/CLI compiler has to generate metadata code like this, where the member functions involved in a property are **specialname**s and a **.property** block connects them as a single abstraction:

```
// ILASM metadata representation
.field private int32 '<backing_store>x'
.method public hidebysig specialname instance int32 get_x() IL managed { ... }
.method public hidebysig specialname instance void set_x(int32 __set_formal) IL managed { ... }
.property instance int32 x() {
  .get instance int32 R::get_x()
  .set instance void R::set_x(int32)
}
```

This requires the compiler to know that apparently independent "get" and "set" functions are related and are intended to form a single property.

The intent of properties is to allow languages and tools to have the syntactic convenience of having non-private member data without the dangers of actually exposing the data directly. For example:

```
obj.x = 42;          // calls set_x
int i = obj.x;       // calls get_x
```

But this is more than just a syntactic convenience: Unlike data members, properties can be virtual (entirely or for certain operations only). Properties are also particularly useful for version safety — the ability to release new versions of a type that are compatible with the earlier version, including the ability to substitute a new version of the type at run time. Exposing an actual data member not only violates data hiding but also ties calling code to that representation and requires a breaking change to make it a function if the programmer later wants to change the representation. Exposing a property, even a default one like the above with simple passthroughs, enables changing the get or set implementation, or even change or remove the backing store, in a version-compatible way that is transparent to calling code using the earlier version of the type.

A different area where properties have become particularly popular is in GUI-based tools which can use type reflection to get a list of an object's properties and then directly expose the ability to edit them to the user of the tool, and changes to the properties correctly execute the appropriate set functions. This usage is popular in many environments besides CLI.

3.2.2 Managed Extensions Design

Managed Extensions did not support properties as a clear language abstraction. Instead, the programmer was required to tag the individual member functions that provide the get and set functionality for the property with a special **__property** keyword and name both functions with a consistent name that correctly follows the pattern **get_...** or **set_...** (and the "…" part of the name has to be exactly the same). The compiler then uses this information to recognize the intent and synthesize the property.

For example, the following class provides a property named **Size**, although it is not clear when reading the code that the compiler silently cobbles together the **get_Size** and **set_Size** functions into a new abstraction:

```
// Managed Extensions syntax
__gc class R {
public:
  virtual void f() { ... }
  __property int get_Size() {              // 1
    return size_;
  }
  __property void set_Size(int size) {  // 2
    size_ = size;
  }
private:
  int size_;
};
```

This design tried to use fewer extensions, but by failing to surface an important abstraction it made things more difficult, not less difficult. It was also brittle and error-prone in practice; for example, if the programmer forgets **__property** one line 1 in the example above then there is still a property but with only a setter, or if the name is misspelled **get_Siz** on line 1 then there are two properties (a read-only one named **Siz** and a write-only one named **Size**). Compilers can add warnings for some of these errors, but in general it's difficult to give high-quality diagnostic messages about cases like this.

3.2.3 C++/CLI Design and Rationale

3.2.3.1 Expose essential differences

The C++/CLI provides an appropriate **property** block abstraction to avoid the duplication and other pitfalls of the earlier design. For example, a property that surfaces an **int** member with just passthrough semantics can be written out as follows:

```
ref class R {
public:
  property int Size {
    int get()              { return size_; }
    void set( int val )  { size_ = val; }
  }
private:
  int size_;
};
```

Incidentally, because CLI library authors frequently expose such a trivial "passthrough wrapper" property (for example to enable future-proofing for version safety even though no special semantics are yet needed), C++/CLI also supports writing the above as a *trivial property*:

```
ref class R {
public:
  property int Size;    // implicitly generates backing store
};                      // and passthrough get and set
```

This form is not only convenient, but lessens the impact on portable C++ code, which in this case could for example **#define property** to be nothing for portability.

Individual get/set functions in the same property can have different accessibilities, can be independently virtual or non-virtual, and can be independently overridden in derived classes. Properties can be static or nonstatic.

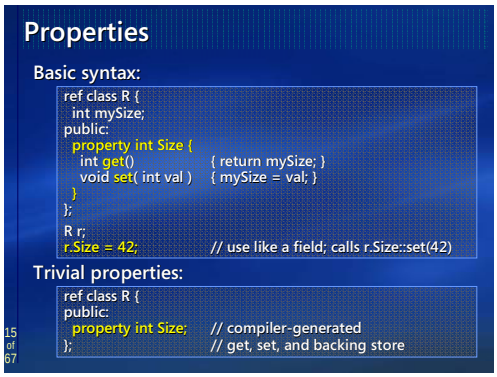


Figure 8: Properties in C++/CLI (from [N1557])

Note that **property** is a contextual keyword that is compatible with all existing C++ code by implementing the "if it can be an identifier, it is" rule. That is, the meaning of existing C++ code like the following that uses **property** as the name of a type or variable is unchanged:

```
property x;          // ok, declares a member of type 'property' named 'x'
int property;        // ok, declares a member of type 'int' named 'property'
```

See also Figure 8, taken from [N1557] slide 15.

3.2.3.2 Future unifications

CLI permits extended language-specific get and set functions besides the usual "same-type" versions. As noted in [N1557], C++/CLI deliberately leaves open the door of in the future allowing overloading a property's **set** function so that a property can be assigned to from other kinds of types. For example:

```
// possible future unification (note: not currently C++/CLI)
ref class R {
public:
  property Foo Bar {
    Foo get();
    void set( Foo );
    void set( int );          // overloaded set function
    template<class T>         // overloaded set function template
    void set( T );
  }
};
```

This would allow uses like:

```
R r;
r.Bar = someFoo;              // call r.Bar::set(Foo) — supported in C++/CLI today
r.Bar = 42;                   // call r.Bar::set(int)
r.Bar = someOtherObject;      // call r.Bar::set<typeof(someOtherObject)>(...)
```

3.2.4 Other Alternatives (Sample)

3.2.4.1 No keyword

One alternative is to use a syntax like the one C++/CLI adopted, but without the **property** keyword:

```
// alternative syntax (note: not C++/CLI)
ref class R {
public:
  int Size {              // note: no "property" keyword
    int get()             { ... }
    void set( int val )   { ... }
  }
};
```

Essentially, this syntax would connote a "data member with get/set functions." This is workable, but seemed to be too subtle. For one thing, the natural syntax for a trivial property would be empty braces, with the following result:

```
// alternative syntax (note: not C++/CLI)
int Size ;              // a data member
int Size { }            // a trivial property
void Size() { }         // an empty function
```

Also, this alternative syntax does not extend as easily to events, which are like properties but have different members (add/remove/raise instead of get/set). Giving the abstraction an explicit name is clearer.

3.2.4.2 property<int>

As already mentioned in §1.2, instead of the chosen syntax:

```
// C++/CLI syntax (trivial property)
property int x;
```

the C++/CLI design could instead have used (among many other alternatives) a library-seeming syntax like the following:

```
// alternative pseudo-library syntax (note: not C++/CLI)
property<int> x;
```

But the difference is only superficial. For one thing, this is still a language feature and a language extension to C++, only now one that pretends to be a library: It requires compiler support and participation because the compiler has to know to emit the **.property** block in metadata.

Further, this option doesn't extend to the common case of user-written get and set functions, because there is no natural place where those could be written in this syntax without breaking the library illusion. For example, either using a special syntax:

```
// alternative not-very-pseudo-library syntax (note: not C++/CLI)
property<int> x { ... ??? ... } // allow a block abstraction anyway?
```

or allowing certain functions to get special handling:

```
// alternative not-very-pseudo-library syntax (note: not C++/CLI)
property<int> x;
int get_x() { ... }             // allow functions that follow a naming pattern and are specially
void set_x( int ) { ... }       // recognized? (see §3.2.2 for some the difficulties with this approach)
```

breaks the illusion and shows that **property<>** is not an ordinary library template.

3.3 CLI Heap (e.g., ^, gcnew)

References were introduced primarily to support operator overloading.

— B. Stroustrup (D&E, p. 86)

C++/CLI adds the **^** declarator for two necessary reasons: to support operator overloading, and to correctly expose the semantics of CLI object references and the CLI heap. It also adds **gcnew** to distinguish the CLI heap clearly from the C++ heap.

A key goal of C++/CLI was to have near-zero impact on C++ programs that just want to gain access to CLI libraries with as little impact to their code as possible, and so syntactic convenience and terseness was a more important consideration for **^** than it was for other any other feature. Note that **^** and **gcnew** are essentially the only features in C++/CLI that are needed to use existing CLI libraries; most other C++/CLI features are only useful for authoring new CLI types and can be ignored by a programmer who is only using existing CLI libraries rather than authoring new ones.

This section discusses the C++ and CLI heaps and focuses on the relationship among the *****, **^**, **new**, and **gcnew** abstractions, as well as the requirements of operator overloading, as a sample of the issues involved with supporting a compacting garbage-collected heap that is not managed by **new**/**delete** or **malloc**/**free**. This section does not mention related topics like references (**&** and **%**, which have an analogous rationale), or **pin_ptr**.

3.3.1 Basic Requirements

There are three basic CLI features that bear on the pointer syntax:

- The nature of the CLI garbage-collected heap.
- The nature of CLI "pointers" (object references) into that heap.
- CLI operator overloading.

The CLI heap is garbage-collected, and allows for implementations to use all major forms of garbage collection techniques — including *compacting collection* where objects can move (change their memory addresses) at any time as the garbage collector moves them together to eliminate memory fragmentation.[6] Importantly, note that CLI garbage collection is permitted to change not only the location of objects in memory, but also their relative address order.

As already noted in §3.1.1, CLI has two major kinds of types:

- **CLI reference types** which can only physically exist on the CLI heap.
- **CLI value types** which have a dual nature, where their "full-fledged object" boxed version can only physically exist on the CLI heap, and their normal "plain blittable value" version can only physically exist on the stack or as directly embedded members of a reference type object on the CLI heap.

CLI also has two kinds of "pointer" types to refer to heap objects. Both are able to *track* moving objects, and so their values can change at any time. This means that they cannot be compared, and cannot be safely cast to another representation that obscures them from the garbage collector (e.g., **int**, or a disk file) and back. They are:

---

[6] Allowing compacting collection is desirable in general because compacting GC has technical benefits, including improving application performance by preserving memory locality with better cache behavior, and allowing fast allocation of arbitrarily sized memory requests.

- *CLI object references* always refer to a *whole object* of *CLI type* on the *CLI heap* (e.g., it cannot refer to a directly held member inside a CLI reference type object, or to an object on the C++ heap). The object reference itself can only physically exist on the stack, in static storage, or on the CLI heap (it cannot physically exist on the C++ heap, although CLI provides a **GCHandle** table that can be used as a mapping to simulate CLI references existing on the C++ heap).
- *CLI interior pointers* can refer to *any object* (including to a value type member directly embedded in a CLI reference object) in *any storage location* (including the C++ heap). But an interior pointer itself can only physically exist on the stack (e.g., it cannot exist on any heap).

That CLI interior pointers can physically exist only on the stack is such a severe limitation that it essentially rules out using this CLI feature under the covers to represent a language-level pointer abstraction. Most of the design discussion will therefore focus on how to deal with CLI object references.

On the other hand, C++ pointers are modeled on memory addresses and are *stable*: They point to objects whose addresses do not change, and whose addresses can be reliably compared. C++ supports direct comparison of pointers into the same object or array, but also comparison of arbitrary pointers via **std::less**. Further, C++ pointers are able to refer to any object in any memory location (although to safely point into the CLI heap requires pinning the CLI object so that it does not move), and C++ pointers themselves can be stored in any memory location.[7]

This creates a tension between CLI object reference and C++ pointers: Although CLI object references are very similar to pointers (e.g., they are an indirection to an object), they are nevertheless semantically not the same because they do not support all the operations that pointers support (e.g., they do not support pointer arithmetic, stable values, or reliable comparison), cannot point to the same things, and cannot be stored in the same places. Some of these differences can be papered over in a semantically sound way that provides consistent abstractions without compromise; other differences cannot be hidden. Pretending that the pointer types are the same, when they are not and cannot be, causes much grief, as was demonstrated by the Managed Extensions.

Table 1 summarizes these characteristics.

In summary, some of these differences can be successfully papered over in the compiler, but some cannot be papered over. CLI object references aren't pointers, and shouldn't be surfaced as such.

The third major CLI feature that bears on the design for a pointer syntax is operator overloading. Like many platforms and languages, CLI provides for overloadable operators. The aspect of CLI operators that is most important to this discussion is that, like all parts of CLI, CLI operators must of course take parameters of CLI reference types via CLI object references.[8] For example:

```
// C++/CLI syntax, to illustrate the parameter passing requirements of CLI operators
ref class R {
public:
  static R^ operator+( R ^r, int i );
  ...
};
```

[7] See also my blog article "Q: Aren't C++ pointers alone enough to "handle" GC? A: No." (November 17, 2003).

[8] There are other aspects of operators that affected the C++/CLI design, notably that the operators must be allowed to be **static** which is also an extension to C++. But unlike the parameter issue above, those other issues don't affect this section's discussion about why C++/CLI represents pointers into the CLI heap as ^.

	C++ pointers	CLI object references	CLI interior pointers
What can it directly point to?	Any object anywhere (but if on the CLI heap the object must be pinned)[9]	Any whole object (not members) on the CLI heap	Any object anywhere
Where can the pointer itself be physically stored?	Anywhere	Anywhere but the C++ heap	On the stack only
Can be reliably compared (i.e., if **p1 < p2** now, it will still be true later)?	Yes	No	No, except to members of the same object or array
Can be stored elsewhere (e.g., cast to **int**, written to disk) and brought back safely?	Yes	No	No

Table 1: Summary of C++ and CLI pointer abstractions

This code is emitted in metadata using a special CLI-reserved name, as something like:

```
.class ... R ... {
  .method public specialname static class R op_Addition( class R r, int32 i ) ... { ... }
  ...
}
```

This has several implications, most notably that whatever way C++/CLI chooses to expose indirection to CLI objects must support operator overloading. Note that C++ had to introduce references (**C&**) as a new form of indirection for similar reasons. As Stroustrup notes:

> *References were introduced primarily to support operator overloading. … C passes every function argument by value, and where passing an object by value would be inefficient or inappropriate the user can pass a pointer. This strategy doesn't work where operator overloading is used. In that case, notational convenience is essential…*
>
> *— B. Stroustrup (D&E, pg. 86)*

In particular, C++ cannot allow overloaded operators that take just C++ pointer parameters, because operators on pointers are already defined by the C++ language; that is, C++ does not let a programmer write the "overloaded operator" signature **operator+(C*, int)**, even if **C** is a user-defined class type.

The same considerations Stroustrup mentions apply also to parameters to overloaded CLI operators:

- Requiring pass-by-value parameters doesn't work, not only because in general it can be inefficient or inappropriate, but specifically it doesn't work for CLI types because most CLI types aren't copyable.

[9] Note that this is a simplification. Pinning is intended to be used for value types (especially **PtrtoStringChars**), so that a non-CLI function can operate on a direct pointer to value type data embedded inside a CLI reference object on the CLI heap.

- Requiring pass-by-***** parameters doesn't work for the same reason it doesn't work in plain C++. (As already noted, CLI object references can't legitimately pull off the masquerade of being ***** pointers anyway. But assuming for the moment that CLI object references were exposed using normal C++ pointer syntax (*****) to pretend they are pointers, that would mean choosing between two undesirable options for overloaded operators: a) disallow natural operator overloading syntax (i.e., the programmer would be required to write **op_Addition** at least at the point of definition, and not be allowed to use **operator+**); or b) create a weird special case to allow operator overloading on **T*** parameters but only for certain types **T**, which seems baroque and would contravene C++'s design of not permitting such overload syntax.)
- Requiring pass-by-**&** parameters doesn't work because C++ references don't have the right semantics to be the right indirection (specifically, they aren't rebindable).

The goal is to let the programmer use natural operator syntax for CLI operators, and write something like the following (where **R?** is a placeholder for some syntax, and **?** is not necessarily a single character):

```
R? operator+( R?, R? );        // R is a CLI reference type
R? r1;
R? r2;
r1 + r2;                       // calls operator+
```

3.3.2 Managed Extensions Design

The Managed Extensions design tried to reduce the number of extensions to C++ by reusing the ***** declarator, where ***** would implicitly mean different things depending on the type that it points to. In the Managed Extensions design, a **T*** is:

- a normal pointer if **T** is a C++ type; or
- a CLI object reference if **T** is a CLI reference type; or
- a CLI interior pointer if **T** is a CLI value type.[10]

But this turned out to be a bad decision, albeit well-motivated to reduce language additions, because it means that **T*** can mean any of three different and incompatible things depending on the type of **T**, and so it obscures an essential difference. For example:

```
// Managed Extensions syntax
template<typename T>                          // if T is a     if T is a      if T is a
void f( T *p ) {                              // C++ type      CLI ref type   CLI value type

  p->SomeValidMemberFunction();               // ok            ok             ok
  p++;                                        // ok            error          ok
  ptrToObjOnCppHeap->someMember = p;          // ok            error          error
  ptrToObjOnCliHeap->someMember = p;          // ok            ok             error
  set<T*> s;                                  // ok            error          error
}
```

[10] This summary suffices to show the difficulties, but it was actually more complicated than this, because this is what you get after applying defaulting rules that simplify things and obscure more complex qualification machinery. The Managed Extensions actually allowed explicitly qualifying pointers with **__gc** or **__nogc** with confusing rules about which one is the default for which kind of pointers to (pointers to pointers to …) particular kinds of types (C++ types, CLI reference types, or CLI value types).

It was an enormous source of confusion to programmers that the Managed Extensions design made something look like a pointer that did not really behave like a pointer, but had different semantics. Tying a type to a particular heap also conflates two concepts that ought to be orthogonal; a storage location ought to be chosen per object, not per type.

Finally, one more significant drawback of the Managed Extensions pointer qualification was the inability to overload operators on CLI classes. CLI libraries define a number of useful overloaded operators, and C++ users clearly wanted to use this functionality with the natural operator syntax. Pointers, however, already have operators defined on them (such as equality, less-than, dereference, and arrow). While it is conceivable that overloading some operators on pointers *to CLI types only* could be done, it was impossible to do so cleanly.

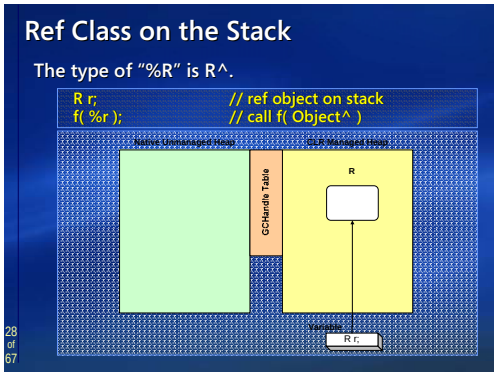


Figure 9: CLI reference type on the stack (from [N1557])

3.3.3 C++/CLI Design and Rationale

3.3.3.1 Paper over differences that can be hidden ("it just works")

Some differences can, and therefore should, be hidden so as to avoid special language rules. For example, it is possible to paper over the difference that CLI reference types are allocated only on the CLI heap.

C++/CLI allows CLI reference types to be instantiated conceptually on the stack and have full stack-based semantics in the language, where the programmer writes something like:

```
R r;                    // R is a CLI reference type
```

and seamlessly gets stack-based semantics even though it is not possible to physically allocate the **R** object on the stack. The compiler physically allocates the CLI reference type on the CLI heap (to satisfy the CLI requirement), and stores an object reference on the stack. See Figure 9, taken from [N1557] slide 28. (See also §3.1.3.)

3.3.3.2 Expose essential differences

The CLI heap is not like the C++ heap, and CLI object references are not like C++ pointers. Above we saw prior attempts that tried and failed to unify the two under a single unextended C++ syntax reusing ***** and **new**. The issue boils down to that we need to surface a CLI abstraction that cannot be expressed in ISO C++ alone, and requires some language support. A key goal of C++/CLI was to find the least intrusive extension.

First, consider CLI object references: They really are not pointers, and can't be papered over to look like them (see §3.3.1 and §3.3.2), and they can't be implemented as a C++ library (see also §3.3.4), so they had to be surfaced as a language abstraction. But they are similar, so they should try to look similar to either normal C++ pointers or C++ smart pointers. What is needed is either an abstraction to express a CLI object reference directly with identical semantics to the CLI feature, or a different abstraction that has different and more desirable semantics but that can correctly hide the CLI semantics under the covers without breaking the language abstraction. C++/CLI chose to surface the feature as-is, as there was no benefit to surfacing it as a different abstraction.

The next question was how to spell it: that is, to find the smallest possible language extension to C++ that could express what was needed and that didn't introduce incompatibility with C++. Reusing the pointer declarator (*****) had already been tried by the Managed Extensions and was known to be a failure, and reus-

ing the reference declarator (**&**) doesn't work because references don't have the right semantics (e.g., aren't rebindable). But to support operator overloading it is highly desirable to expose object references under some kind of a pointer-like declarator, so a declarator was needed.

Stan Lippman was the first to strongly make the case why a new abstraction was needed and why it should use a new declarator that was not already used by C++ (and that was unlikely ever be used in C++0x evolution, so as to avoid interfering with design choices C++ might make in the future). The set of such available declarator symbols is fairly small, and Stan suggested **^**, which was adopted after some flirtations with other symbols because **^** seemed to most clearly connote a pointer. (It was not chosen because **^** was used by Pascal, although Pascal programmers may find it familiar.) Writing an overloaded CLI operator therefore can use natural C++ operator declaration syntax instead of something jarringly different:

```
R^ operator+( R^, R^ );       // R is a CLI reference type
R^ r1 = ...;
R^ r2 = ...;
r1 + r2;                      // calls operator+
```

Another reason to use a declarator like **^** is that this would be the most frequently used extension in all of C++/CLI, so also for that reason it made sense to choose the terser and arguably more natural syntax that also did not pretend to be something it was not, had the cleanest correspondence to pointers both operationally and visually, and most cleanly supported operator overloading.[11]

At this point it is important not to fall into the "**^** is dereferenced with **^**" pitfall. The operations you can perform on a **^** (e.g., dereferencing, indirection) are a subset of the operations that are legal on a *****, and those operations they have in common should be spelled the same way in order to avoid needless divergence and to support writing templates that can deal with either ***** or **^**. For example, just as we can dereference a ***** using the **->** operator, we should be able to dereference a **^** using the **->** operator too:

```
T* p = ... ;
p->f();                 // call T::f
T^ h = ... ;
h->f();                 // call T::f
```

The pitfall is that it is tempting to say that, just as a pointer declared using ***** is dereferenced using the unary ***** operator, therefore "naturally a **^** should be dereferenced using the unary **^** operator," for example:

```
// note: not C++/CLI
T* p = ... ;
(*p).f();
T^ h = ... ;
(^h).f();               // the pitfall — not a good idea
```

I recall a design session where Bjarne Stroustrup and I were working in his office one afternoon to work through alternative syntaxes and semantics for exposing the CLI object reference abstraction. At first, the

---

[11] What about CLI interior pointers? Those are used rarely, and can be more expensive to overall performance because the garbage collector has to do more complex accounting to deal with the fact that they could point inside a CLI heap object (rather than a whole object, which is easier to track by comparing just start addresses); that is one reason they are only allowed to exist on the stack, to limit their number and lifetime. Therefore it makes sense to surface them as a library-like syntax **cli::interior_ptr<T>**, deliberately making a feature that is less usual also look less usual. For details on what this means in the implementation and why it's still a language feature, see the issues covered in the discussion of the **handle<T>** alternative in §3.3.4, which apply also to **cli::interior_ptr**.

foregoing seemed reasonable. The next time we spoke, however, Bjarne suggested changing the design so that ^ would be dereferenced instead with the unary * operator, just like pointers, because when he had first sat down to write a template that could use either form the problem became evident:

```
// Stroustrup's counterexample (note: not C++/CLI)
template<typename SomePtrType>
void f( SomePtrType p ) {
  p->f() ;            // ok, works whether SomePtrType is a *, ^, or smart pointer type
  (*p).f();           // error if SomePtrType is a ^ (ok if it's a * or smart pointer type)
  (^p).f();           // error if SomePtrType is a * or smart pointer type (ok if it's a ^)
  delete p;           // ok, works whether SomePtrType is a *, ^, or smart pointer type
}
```

This made the issue clear: The unary * operator applies more generally than just to the * declarator, even though both happen to be spelled using the same symbol in C. Specifically, unary * (along with related operations like -> and **delete**) forms part of the algebra of legal operations on any pointerlike object, which in C++ already included both normal pointers and smart pointers.[12] So the correct design is what Stroustrup suggested and what C++/CLI adopted:

```
// corrected, and now C++/CLI
T* p = ... ;
(*p).f();             // dereference T* with unary *
T^ h = ... ;
(*h).f();             // dereference T^ with unary * — writing "(^h).f();" is an error, no unary ^
```

This allowed writing agnostic templates that didn't care whether they were given a *, a ^, or a smart pointer — as long as they limited themselves to the subset of operations that were supported on all three, they were spelled the same way in each case:

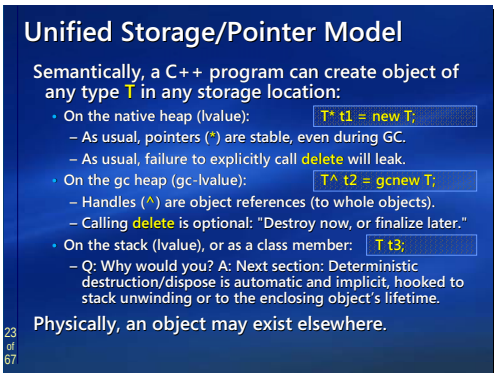
```
// Stroustrup's example now supported in C++/CLI
template<typename SomePtrType>
void f( SomePtrType p ) {
  p->f() ;            // ok, works whether SomePtrType is a *, ^, or smart pointer type
  (*p).f();           // ok, works whether SomePtrType is a *, ^, or smart pointer type
  delete p;           // ok, works whether SomePtrType is a *, ^, or smart pointer type
}
```

C++/CLI now had to resist the temptation to "support unary ^ anyway for symmetry" with the same meaning as unary *. For one thing, that wouldn't add any expressive power: Programmers could already use unary *. For another, in general I feel it's a bad thing to introduce two equivalent ways of doing something when there's no good reason to do so; besides, it's always good to leave doors open for the future wherever possible, and it could be that someone may discover a useful and compelling meaning for unary ^ that is not known today. Experience has shown that preserving C++'s rule that "-> and unary * dereference any pointerlike thing" is easy to for programmers to learn, and typically those programmers who do try to dereference ^ using unary ^ in their first C++/CLI program just find that it doesn't work, switch to using unary * instead, and never notice it again.

---

[12] Put another way: There's conceptually a supertype called "pointer-like object" with operations *, ->, and **delete**, and there are subtypes for traditional pointers, smart pointers, and CLI references. These subtypes have their own syntax for declaration (e.g., **T***, **smart_ptr<T>**, and **T^**) and creation (e.g., **new T**, **gcnew T**), but then can be used as-a "pointer-like object."

For a summary of the heap and pointer model, see Figure 10, taken from [N1557] slides 23 and 24.



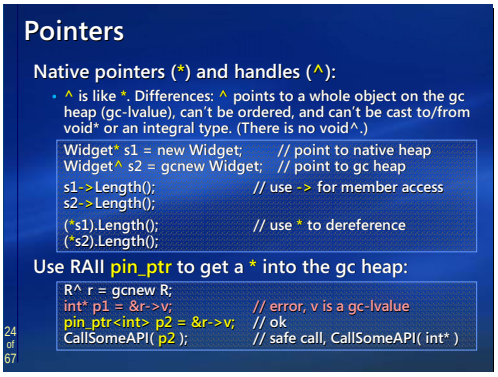


Figure 10: Summary of the unified heap and pointer model in C++/CLI (from [N1557])

The other major question was how to surface the difference that the CLI heap was different from the C++ heap. One possibility would have been to just use **new**, as in:

```
// a rejected alternative syntax (note: not C++/CLI)
R^ r = new R;                    // R is a CLI reference type
```

The idea is that, if the type is a CLI type, the compiler implicitly decides to allocate it on the CLI heap. This is what the Managed Extensions did, and it has two important shortcomings:

- **It fails to treat types and heaps orthogonally, but bind certain types implicitly to certain heaps.** This lack of orthogonality closes the door to the future simplification of allowing any type to be allocated conceptually on the C++ heap with C++ heap semantics (and so make them directly usable to existing C++ template libraries that use only ***** and **new**) and vice versa. See the next subsection for discussion about these future simplifications. Rather than saying that certain types are always on certain heaps, the CLI heap should be a first-class abstraction.
- **It makes the definition of new problematic.** In C++ a new-expression **new T** always has the type **T*** (not **T^**) and it calls some **operator new** which returns a ***** (not a **^**). It's very important to be able to say crisply what something's type is, and so I considered that it would be unusual and arcane to meddle with the specification of the type of something as fundamental as **new T** and **operator new** by adopting a rule that the type of **new T** is sometimes a ***** and sometimes a **^**, and likewise that **operator new** should sometimes be written to return a ***** and sometimes to return a **^**. It seemed to me that such a lack of clarity about the type of the **new**s amounted to intrusive monkeying with a basic feature, and that it would be prone to get in the way of ISO C++ and its evolution.

So what was needed was something like **new**, but distinct from it, to represent allocation the CLI heap. If not **new**, then what? C++ already supports extensions to **new** using placement **new**, so one alternative was to use that, for example:

```
// other rejected syntaxes for "gcnew" (note: not C++/CLI)
R^ r = new (gc) R;
R^ r2 = new (cli) R;
```

Of course, one issue is that it has similar problem about "what is the type of **new**" as the alternative just discussed. But this choice could additionally have conflicted with C++0x evolution which might want to provide additional forms of placement **new**, and of course using a placement syntax could and would also conflict with existing code that already uses these forms of placement **new** — in particular, **new (gc)** is already used with the popular Boehm conservative garbage collector for the C++ heap. After all, a key goal of C++/CLI is to support all ISO C++ programs with unchanged semantics, and that includes any use of optional garbage collection for the C++ heap.

So C++/CLI chose **gcnew** to go with ^, and the result looks like:

```
T* p = new T;        // allocate on C++ heap
T^ h = gcnew T;      // allocate on CLI heap
```

In retrospect this should probably have been spelled **gc new** to avoid taking a reserved word here. Alternatively, **cli new** might have been a slightly better choice, with less connotation that the CLI heap might be the only garbage-collected heap (which is not necessarily true today as some C++ implementations support conservative garbage collection for the C++ heap, and especially not true if C++0x adds explicit support for garbage collection in the future).

Finally, note that **delete** does not need a special treatment. Consider that **delete** is just one of the set of valid operations on pointers and handles, just as unary * and -> are. We can and therefore should make **delete** "just work" with the usual meaning of calling the destructor, regardless of the type of pointerlike thing it is given (* or ^).

3.3.3.3 Future unifications

There are two other notable semantics C++/CLI could support in the future without any new language extensions, but that can be added in a way that can be completely papered over by a compiler:

- Semantically allocating a CLI object on the C++ heap using **new** with C++ heap semantics (including no finalizers and no guaranteed garbage collection; manual memory management with explicit **delete** is required to avoid leaks) and pointing to it with a C++ pointer.
- Semantically allocating a C++ object on the CLI heap using **gcnew** with CLI heap semantics (including finalizers and guaranteed garbage collection) and pointing to it with a C++/CLI handle.

These were deferred from C++/CLI to manage project scope, and C++/CLI does not support these features now; it just deliberately leaves the door open for adding them in the future if there is ever a second edition of C++/CLI.

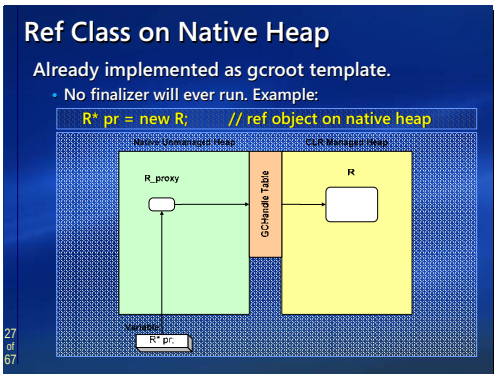


Figure 11: CLI reference type on the C++ heap (from [N1557])

Here is how to paper this over: To allow CLI reference types to have C++ heap-based semantics in the language, the idea is to let the programmer write something like:

```
R* r = new R;        // R is a CLI reference type
```

and seamlessly get C++ heap-based semantics (including the need to explicitly **delete** the object, and the ability to leak it by failing to call **delete**) even though it is not possible to physically allocate the **R** object on the C++ heap. To implement this, the compiler physically allocates the CLI reference object on the CLI heap (to satisfy the CLI requirement), and allocates a proxy object on the C++ heap that contains a **GCHandle** table entry to the actual **R** object.[13] See Figure 11, taken from [N1557] slide 27.[14]

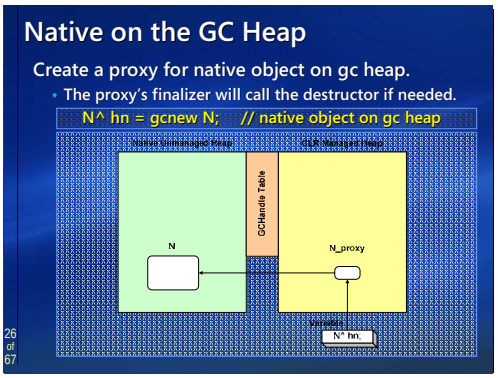


Figure 12: C++ type on the CLI heap (from [N1557])

[13] This is just a simple sketch. There are many other issues a complete design would need to consider, such as to consider allowing an implicit conversion from **R*** to **R^** (where **R** is a CLI reference type) so that CLI functions can be called naturally

Similarly, to allow C++ types to have CLI heap-based semantics in the language, the idea is to let the programmer write something like:

```
C^ c = gcnew C;      // C is a C++ type
```

and seamlessly get CLI heap-based semantics even though it is not possible to physically allocate the **C** object on the CLI heap. To implement this, the compiler physically allocates the C++ object on the C++ heap as usual, and allocates a CLI proxy object on the CLI heap that contains a pointer to the actual **C** object. See Figure 12, taken from [N1557] slide 26.

It's worth noting that, earlier in the C++/CLI design effort, some people tried hard to leave open the possibility of allocating C++ objects *physically* on the CLI heap; that is impossible in general, and trying to persist in this led to confusions in the type system that have now been avoided and removed. The reason why it is impossible to ever place C++ objects directly on the CLI heap is because C++ pointers (including **this**) can't point to things that move, and in general C++ objects can and do have their addresses taken, store or expose pointers to their members, store or expose copies of their **this** pointers, and so on. At best, putting a C++ object physically on the CLI heap would require pinning it for its lifetime, and that is completely unworkable (see the alternative of "pin everything" in §3.3.4). The right approach is to stick to the clarifying simplification that each kind of object physically lives on its own heap in the implementation, and allow putting it logically and semantically on the other heap in the programming model (as shown above).

Finally, note that although this papering over can be done in a way that preserves correct and consistent semantics without holes or pitfalls, just like with virtual function calls the extra indirection is not free: Doing this incurs a double indirection on all C++ "pointers" to CLI objects, and vice versa. But it does offer the convenience of using CLI types seamlessly and correctly with existing C++ libraries (e.g., container template libraries) that use only **new** and *****, and that do take care to **delete** C++ heap objects.

3.3.4 Other Alternatives (Sample)

3.3.4.1 Double indirection

A variant of the Managed Extensions design would be to use **T*** for everything, but if **T** is a CLI type then add an indirection: Make it a real C++ pointer to a nonmoving proxy on the C++ heap that contains a **GCHandle** to the actual object.

This is similar to Figure 11, with one important difference: If this were the *only* way to use CLI objects, it would be unusable in common situations. Specifically, it would create these two serious issues:

- Every use of a CLI object would require a double indirection. This would often be unacceptable in performance-sensitive code due to poor locality (e.g., due to the indirection itself, and due to cache effects).
- Every CLI object used in a C++ program would be required to obey C++ heap semantics only, including manual memory management. CLI objects created in a C++ program would not enjoy garbage collection, and would have to be explicitly deleted or else be leaked.[15] It is desirable to have

(e.g., **SomeCliAPI(r)**) without ugly contortions to convert an **R*** to an **R^** by dereferencing the ***** and applying unary **%** to get a **^** (e.g., **SomeCliAPI(%*r)**).

[14] Something analogous would also be done for CLI value types, but note that the correct design for supporting **V* v = new V;**, where **V** can be any kind of value type, is nontrivial and not just an obvious extension of the design for reference types. In particular, a correct design for **new** of value types and pointers to value types must account for: a) subtle issues arising from value types' dual nature (i.e., boxed and unboxed); and b) unlike reference objects, some value objects can exist physically on the C++ heap (e.g., **Int32** which is a synonym for **int**).

[15] Specifically, the proxy on the C++ heap keeps the CLI object on the CLI heap alive and so governs the CLI object's lifetime.

these C++ heap semantics available; but it is not desirable to have them be required all the time and inescapable.

That is why, although the previous section showed why allowing new of a CLI type is desirable for convenience where performance is not critical and C++ heap semantics are acceptable (e.g., to let existing C++ code and libraries that use simply **new** and ***** to work with CLI types without change), it is not viable to make that the only way to use CLI types and would have prevented achieving the goal of enabling C++ to be a viable systems programming language on CLI.

3.3.4.2 Fat pointers

Another variant of the Managed Extensions design would be to use **T*** for everything, but if **T** is a CLI type then store more complex information. For example, a **T*** to a CLI type **T** could store both a CLI object reference and a CLI interior pointer, as an attempt to let it point at anything; that particular example fails to even get out of the gate, however, because such a structure that contains a CLI interior pointer can then only exist on the stack which would be even worse at supporting the C++ ***** pointer abstraction.

In general, alternatives based on pointers with extra information lead to subtle surprises, such as that **sizeof(T*)** is different depending on the type of **T**, and that pointer compare-and-swap operations are no longer easy to make atomic which is generally unacceptable for concurrent code. The obvious way to get around that, in turn, is for the compiler to actually represent the pointer under the covers using an extra indirection (e.g., every pointer to a reference type is actually a normal pointer to a fat pointer containing actual pointer information about the object), which is a variant of the previous alternative with the same drawbacks.

3.3.4.3 Pin everything

One frequently suggested option would be to pin every object on the CLI heap so that pointers could just refer to CLI heap objects directly and no new abstraction would be necessary. This is completely unworkable, because pins interfere with garbage collection and therefore can only be used sparingly and for brief periods, such as keeping a pin for the duration of a C++ function call when you need to pass a pointer to a CLI heap object to a C++ function that requires a real pointer (and of course that function must not try to keep a copy of the pointer because the pin will be released after the function call returns). Conceptually, a pin is a "sandbar" that a compacting garbage collection pass cannot cross; in pathological cases, a pin can in theory prevent memory allocation even when memory is available by preventing collection and defragmentation (e.g., pinning an object very near the end of the CLI heap address space and then trying to allocate CLI heap memory can fail even when recoverable memory exists beyond the pinned object). So pinning everything, which effectively means turning off garbage collection and in general never freeing any CLI heap objects, isn't an option.

3.3.4.4 handle<T>

An obvious alternative is to do basically what C++/CLI did, but call it **handle<T>** instead of **T^**. Note that this is not making it a library; it is still a language feature, only dressed in library-like sheepskin (see §1.2). Consider that the compiler still needs to know to emit an object reference, and so has these major choices:

- **Recognize handle<T> as a special type,** the same way that C++/CLI treats **cli::array** and **cli::pin_ptr**: If name lookup finds the marker type in namespace **cli**, have the compiler treat it with special meaning. This pretends that conceptually it is a "library type," even though none of these are real library types because their implementations are provided by the CLI runtime environment and the names are hardwired into the C++/CLI compiler to be specially recognized as tags for those CLI runtime facilities.

- **Implement handle<T> as a real type but with a magical implementation.** This could also have been done, but doesn't remove the need to add a language extension, because the internals of **handle<T>** still can't be written without having a language extension to surface CLI object references. (Note that C++/CLI enables programmers to trivially write such a **handle<T>** as a passthrough wrapper library template if they want to wrap the C++/CLI-specific **^** for better portability, but the language feature is needed for the library wrapper type to use internally in its implementation.)

So **handle<T>** is still a language feature, since its implementation requires some form of language support. Given that both **handle<T>** and **T^** are equally language extensions, C++/CLI chose the extension with terser and arguably more natural syntax, especially considering that this was going to be by far the most frequently used feature of all of C++/CLI; programmers who only want to use CLI libraries might never use more than **^** and **gcnew**. Requiring a **handle<T>** pseudo-library language syntax would force programmers to use a more tedious syntax for a common feature without the actual benefit of really avoiding a language extension.[16]

[16] For additional discussion about the **^** design and alternatives that were explored, see also Brandon Bray's blog entry "Behind the Design: Handles" (November 17, 2003).

3.4 CLI Generics (generic)

C++/CLI adds the **generic** contextual keyword because CLI generics are different from templates, and because it is necessary for the compiler to recognize CLI generics in order to generate correct metadata. Further, C++/CLI cleanly integrates the two kinds of type genericity so that they can work powerfully together and each can be used for its strengths, alone or in combination. This section avoids repeating points already made earlier in this paper about when and why language support is required, and focuses on the design for presenting this specific feature in the programming model.

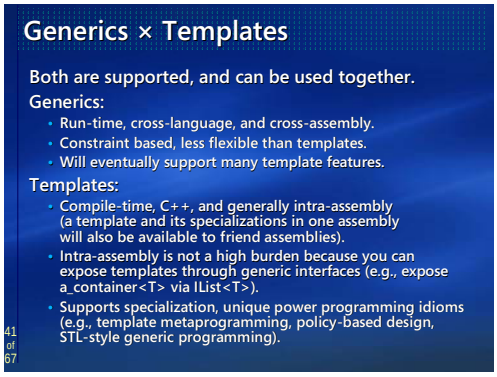


Figure 13: Similarities and differences between templates and generics (from [N1557])

3.4.1 Basic Requirements

CLI allows generic types and member functions that are parameterized by types. Although both templates and generics enable forms of type genericity, generics are different from templates in many ways, including those summarized in Table 2.

	C++ templates	**CLI generics**
When and where instantiated	Compile time, intra-assembly	Run time, cross-assembly
How and when type checked	Partly separately in advance at the point where the template is defined, and partly for each instantiation at instantiation time	Always separately in advance at the point where the generic is defined, for all possible instantiations
Understood by	C++ only	All CLI languages (required for CLS consumers)
Can be specialized	Yes	No
Can have default type parameters	Yes	No
Can have non-type parameters	Yes	No

Table 2: Summary of some major features of C++ templates and CLI generics

Both have desirable behaviors, but they are not the same. (See Figure 13, taken from [N1557], slide 41.) Nevertheless, the two are closely enough related that any syntax for generics ought not to be surprisingly different from C++ templates, and in their area of feature overlap they ought to be able to be used together.

3.4.2 Managed Extensions Design

Not applicable: The Managed Extensions design supports the first edition of ISO CLI, which predates generics.

3.4.3 C++/CLI Design and Rationale

Generics cannot be exposed using **template<>** syntax because they are not templates and do not behave the same way. Besides, it is legal and useful in C++/CLI to write both templates and generics of CLI types, and so they must be expressed as orthogonal features with distinct syntax.

Because generics have a similar basic function, however, they ought not to be exposed with a jarringly different syntax from templates (which would also prevent their compatible use; see below). Rather, the natural syntax that most programmers expect to find is the one C++/CLI chose:

```
generic<typename T>                    // "<typename T>" or "<class T>", as with templates
ref class R { ... };
```

To preserve symmetry with C++, **typename T** and **class T** can be used to declare generic type parameters.

Because both generics and templates support type genericity, however, they should be well integrated in their area of overlap. In particular:

- Programmers should be able to write templates that can instantiate both templates and generics.
- Therefore generics ought to be able to match template template parameters.
- Templates should be able to inherit from generics, especially from generic interfaces.

For example:

```
template< template<class> class X >
void f() {
  X<int> x; // instantiate X (note: one instantiation syntax)
  // ... use x ...
}
generic<typename T> ref class GR { ... };
template<typename T> class TC { ... };
f<GR>();    // ok
f<TC>();    // ok
```

This works, and it enables powerful idioms. For example, consider an STL-style **vector** template that additionally implements a generic CLI **List<T>** interface: The template can be instantiated in C++ code, and then the object can be seamlessly passed to CLI code written in an arbitrary CLI language that can traverse it naturally using the **List<T>** interface, and other languages that provide "foreach"-like language loop constructs that recognize well-known CLI interfaces can use these language loop constructs seamlessly on the STL-style vector template instantiation. (See also Figure 14, taken from [N1557], slides 42 and 43.)

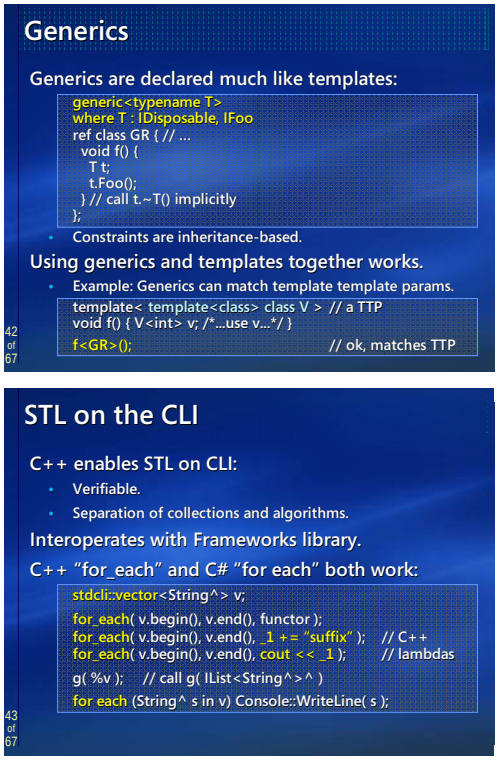


Figure 14: C++/CLI support for generics (from [N1557])

3.4.4 Other Alternatives (Sample)

3.4.4.1 Reuse template syntax

One option would have been to reuse **template** declarations to implicitly mean generics on CLI types, and disallow template-only features despite the **template** keyword:

```
template<typename T> ref class R { ... };     // generic?
```

This would be wrong because generics do not behave like templates (e.g., they cannot be specialized), so this attempts to give an illusion that is both wrong and will quickly become visible to the programmer.

But this alternative immediately fails for a different reason: It is not orthogonal, and would prevent actually allowing templates of CLI types — which C++/CLI does in fact support (see §3.1.3). The above code is legal C++/CLI, but means that **R** really is a template whose instantiations are CLI reference types.

3.5 C++ Features (e.g., template, const)

A key goal of C++/CLI is orthogonality: To allow C++ features to be used also on CLI types and vice versa, so that programmers would not have to remember a quagmire of rules of the form "feature X works only on CLI types" and "feature Y works only on C++ types." Some restrictions of that form are necessary, and some are not necessary but did not make the cut for the initial release of C++/CLI — but many restrictions of that form can and have been eliminated in C++/CLI.

The issue is not compiling C++ code to target CLI, for C++/CLI simply supports all of ISO C++ unchanged and so all C++ features work as usual on C++ types. (Compiling C++ code to target CLI is described in §2.1.) Rather, the issue here is imbuing CLI types with extended C++ language-specific semantics.

Although CLI has no inherent support for several important C++ features, including templates and **const**, C++/CLI nevertheless has been able to add support for these features, and it is useful to understand when and how they apply (or don't), as well as doors that have been deliberately left open for future unifications. This section focuses on the **template** and **const** features as two cases in point to enable a discussion about the issues involved with supporting C++ features also for CLI types that have no inherent knowledge or support for them.

3.5.1 Basic Requirements

CLI is designed to support many languages and compilers. It therefore has explicit support for adding modifiers that mark members and function parameters having different semantics from CLI, and C++/CLI makes use of this latitude. There are two important kinds of modifiers:

- **modopt:** A "optional modifier" that marks a type or function as having some special semantics beyond those recognized by CLI, where a CLI consumer language that does not understand the modopt can still use the function correctly.
- **modreq:** A "required modifier" that marks a type or function as having some special semantics beyond those recognized by CLI, where a CLI consumer language is required to either understand and support the modreq or not allow calling the function. (From the viewpoint of code written in that language, the effect usually is as though the function didn't exist; it simply can't be used correctly without knowing essential required semantics.)

Modopts and modreqs are significant in function signatures. In particular parameter types differing only in modopts or modreqs are considered different for overloading. See [C++/CLI §33] for further information about modreqs and modopts.

3.5.2 Managed Extensions Design

3.5.2.1 Templates

The Managed Extensions design did not support any interaction between templates and CLI types: It did not allow templates of CLI types, member templates of CLI types, or instantiating templates with CLI types.

```
// Managed Extensions restrictions
class C { };
__gc class R { };
template<typename T> class TC { };
T<C> tc;                                  // ok
T<R> tr;                                  // error
template<typename T> __gc class TR { } ;  // error
```

3.5.2.2 const

The Managed Extensions design did not allow **const** member functions on a CLI type. It did, however, allow a CLI type to have **const** data members. For example:

```
__gc class R1 {
public:
  void f() const;                          // error
};

__gc class R2 {
  const int x;                             // ok
public:
  R() : x( ... ) { }                       // initialize x
};
```

3.5.3 C++/CLI Design and Rationale

3.5.3.1 Templates

Templates are fully integrated with CLI types and generics and "just work." From [C++/CLI §30]:

> *The template syntax is the same for all types, including CLI class types. Templates on CLI class types can be partially specialized, fully specialized, and non-type parameters of any type (subject to all the constant-expression and type rules in the C++ Standard) can be used, with the same semantics as specified by the C++ Standard.*
>
> *Templates are fully resolved and compiled at compile time, and reside in their own assemblies.*
>
> *Within an assembly, templates are implicitly instantiated only for the uses of that template within the assembly.*

As described in §3.1.3, C++/CLI allows templates of CLI types, templates instantiated with CLI types, argument type deduction for function templates, and all other template features. For a basic example:

```
class C { };
ref class R { };

template<typename T> class TC { };
T<C> tc;                                   // ok
T<R> tr;                                   // ok
template<typename T> ref class TR { } ;    // ok
```

Like all templates, CLI type templates are instantiated by the compiler at compile time, and for each instantiation the compiler generates a CLI reference type with a suitably mangled unique name. Templates can only be instantiated in the same assembly, but an object of the instantiated types can be passed to code written in any CLI language (after all, at that point it is just a regular CLI object whose type has a funny name).

Further, as described in §3.4.3, C++/CLI supports a high degree of integration between templates and CLI generics. For example, CLI generics can match C++ template template parameters, and C++ templates can implement CLI generic interfaces or inherit from CLI generic types. For example:

```
generic<typename T>
interface class List { ... };              // List is usable by other CLI languages

template<typename T, typename A = std::allocator<T>>
ref class vector : List<T> { ... };    // vector can be specialized, use default template parameters, etc.
```

This has enabled important idioms such as an extended implementation of STL whose container types are templates and so can only be instantiated in C++, but once instantiated the container objects can be passed to and naturally used by other CLI code even if that code is written in other languages because the containers implement well-known CLI generic interfaces. (See Figure 14, slide 43.)

A CLI type can also have a member function template. For example, consider this code that defines a CLI reference type R having a member function template **f**:

```
ref class R {
public:
  template<typename T>
  void f( T t ) { }
};
```

Here **R::f** can be instantiated to take a parameter of any kind of type, with usual template type deduction in deducible contexts. For example:

```
class C { };

int main() {
  R r;
  C c;

  r.f( 42 );          // instantiate with fundamental type (deduce T = int)

  r.f( c );           // instantiate with C++ type (deduce T = C)

  r.f( gcnew R );     // instantiate with CLI type (deduce T = R^)
}
```

For the above code, the following ILASM shows the instantiations that the compiler generates for the different types and how the names are reflected in metadata:

```
.class private auto ansi beforefieldinit R extends [mscorlib]System.Object {
  .method public hidebysig instance void 'f<int>'(int32 t) cil managed { ... }
  .method public hidebysig instance void 'f<C>'(valuetype C t) cil managed { ... }
  .method public hidebysig instance void 'f<R ^>'(class R t) cil managed { ... }
  .method public hidebysig specialname rtspecialname instance void  .ctor() cil managed { ... }
}
```

A contemplated future extension is to allow property set functions to be templates (see §3.2.3)

3.5.3.2 const

Like the Managed Extensions, C++/CLI does not yet allow **const** member functions on a CLI type. For example:

```
ref class R1 {
public:
  void f() const;                          // error
};
```

But member functions of a CLI type can have **const** and **volatile** parameters, and CLI types are allowed to have **const** and **volatile** data members. For example:

```
ref class R2 {
  const int x;                          // ok
public:
  R() : x( ... ) { }                    // initialize x
  const int f( const int* );            // ok
};
```

As described in [C++/CLI §33.1.2]:

> *The distinction between required and optional modifiers is important to tools (such as compilers) that deal with metadata. A required modifier indicates that there is a special semantic to the modified item, which shall not be ignored, while an optional modifier can simply be ignored. For example, **volatile**-qualified data members shall be marked with the **IsVolatile** modreq. The presence of this modifier cannot be ignored, as all accesses of such members shall involve the use of the **volatile.** prefixed instruction (see §33.1.5.9 for an example). On the other hand, the **const** qualifier can be modelled with a modopt since a **const**-qualified data member or a parameter that is a pointer to a **const**-qualified object, requires no special treatment.*
>
> *The CLI itself treats required and optional modifiers in the same manner.*

For example, from [C++/CLI §33.1.1]:

> *Consider the following class definition:*
>
> ```
> public ref class X {
> public:
>   static void F(int* p1) { ... }
>   static void F(const int* p2) { ... }
> private:
>   static int* p3;
>   static const int* p4;
> };
> ```
>
> *The signatures of these four members are recorded in metadata as follows:*
>
> ```
> .method public static void F(int32* p1) ... { ... }
> .method public static void F(int32 modopt([mscorlib]System.Runtime.CompilerServices.IsConst)* p2) ... { ... }
> .field private static int32* p3
> .field private static int32 modopt([mscorlib]System.Runtime.CompilerServices.IsConst)* p4
> ```

So **const** is supported for CLI types with only minor limitations,[17] except primarily for **const** instances of CLI types and **const** member functions on CLI types. C++/CLI deliberately leaves the door open to support **const** instances and **const** member functions on CLI types in the future, but note:

- The design for encoding **const** member functions in metadata is not trivial; for example, CLI does not allow putting a modopt or modreq directly on a function. As of this writing, I can think of three major alternative techniques, each of which would need to be considered in detail to ensure that it works correctly in all cases and don't unwittingly close doors or cause unwanted interactions.
- Even when **const** instances and functions are supported, they can only be applicable to types both authored *and* consumed using a C++/CLI compiler. For example, permitting a **const** instance is useless for all existing CLI types and libraries, because existing libraries have no **const** functions; allowing a **const** object of such a type means allowing an object you can't do anything with, because it has no callable member functions.

[17] For example, the current encoding could be improved to distinguish between **const*** and ***const**.

4 Some FAQs

Why create a technology like C++/CLI?

To ensure that C++ developers would not be shut out from an important platform, ISO CLI (the leading commercial implementation of which is Microsoft's .NET).

C++/CLI's mission is to provide direct access for C++ programmers to use existing CLI libraries and create new ones, with little or no performance overhead, with the minimum amount of extra notation, and with full ISO C++ compatibility.

Why is support for CLI important?

Before joining Microsoft in 2002, I had mostly ignored CLI and .NET and wasn't very concerned about whether C++ supported them well. Then I saw the extensive development efforts and huge investments going on for WinFX and Longhorn (now called Windows Vista), and it became clear to me that any language that did not support CLI programming well would increasingly be marginalized for programming on the Windows platform as a whole.

CLI libraries are the basis for many of the new technologies and libraries on the Windows platform, including the rich (i.e., huge) WinFX class library shipping with Windows Vista which offers over 10,000 CLI classes for everything from web service programming (Communications Foundation, WCF) to the new 3D graphics subsystem (Presentation Foundation, WPF). Languages that do not support CLI programming have no direct access to many of these important libraries and are shut out from them. Programmers who want to use those ever-richer features, either because of the libraries' inherent usefulness or just to remain competitive in look-and-feel with other software and with Windows itself on each new latest release of the operating system, are forced to switch to one of the 20 or so other languages that do support CLI development, at least for the parts of their applications that use those features, even if they would have preferred to continue using C++. Languages that support CLI include COBOL, C#, Eiffel, Java, Mercury, Perl, Python, and others; at least two of these have standardized language-level bindings.

Isn't there lots of non-.NET development still taking place, even on Windows?

Yes. Of course C++ would not immediately die away completely on Windows even without good CLI support, because there is still a place for non-.NET Windows development. But C++ would be shut out from most major new features, and that is not a long-term tenable position; more and more programmers would be forced to use other languages over time. In the view of many developers, C++ had already been marginalized as "legacy only," and I believe that was something that needed to be fixed. Conversely, I realized that C++'s valuable strengths as a programming language (e.g., templates, destructors) would be just as important and useful for CLI programming as they have been for programming on other platforms, and that their absence would be a real loss for CLI developers.

Does CLI support matter for C++ programmers who are not on Windows?

Not as urgently. Implementations of ISO CLI are available for four major operating systems other than Windows, but today most C++ programmers on other platforms do not need to use CLI libraries and they can ignore C++/CLI. The indirect benefit for the C++ community as a whole is that keeping C++ relevant on an important platform is good for C++ use in general, by enabling programmers who prefer to stay with C++ to do so.

Don't language extensions fracture the C++ community with dialects?

That is especially a risk with incompatible extensions that compete with the standard, and this is best avoided by sticking to conforming extensions that scrupulously maintain ongoing compatibility with the standardized core.

All C++ compilers add language and library extensions for system-specific details, and each compiler adds different ones — this is in itself normal, expected, and when done correctly it is standards-conforming and doesn't create competing dialects. The ISO C++ standard is the unifying force that has prevented fracturing of the C++ marketplace while specifically allowing for such compiler extensions. The threat of competing dialects arises most often in the following cases, all of which exhibit some kind of incompatibility with (or absence of) a standardized core:

- **Incompatible variation without an accepted standardized core.** This was common particularly in the first half of the 1990s, when different pre-Standard C++ compilers used to support features like templates under the same syntax but with different semantics.
- **Incompatible extensions that contradict the standardized core.** This means that standard-conforming code breaks or has a different meaning under the extensions, which causes fragmentation by forcing users to choose between the standard meaning or the competing dialect's meaning.
- **Attempts to define subsets of the standardized core.** These are by definition incompatible, and cause fragmentation by forcing users to choose between using the entire standard or only the features in the competing subset specification.

C++ has largely avoided any serious fragmentation because C++ compiler vendors have tended to be careful to support the entire standard, and to use pure extensions that do not change the meaning of the standardized core and furthermore do not interfere with C++ language and library evolution. C++/CLI bends over backwards to be fully compatible with ISO C++ and with C++0x evolution, which it does better than any set of extensions to any programming language, standard or otherwise, that I know of.

The Managed Extensions already existed. Wasn't that a solution?

No. The Managed Extensions design was a valiant and well-intentioned effort, but it had serious flaws (both technical and aesthetic) that made it not usable in practice. As a result, it was widely rejected by programmers. It specified fewer extensions to C++ than C++/CLI does, and that was a part of the problem; its frugality often obscured essential differences and made it confusing to understand and use.

Why try to standardize C++/CLI? Isn't it enough that C++/CLI exists?

Having a standard enables people to create independent, compatible implementations. It also promotes participation and input from the community, and C++/CLI has already benefited from community input.

CLI itself is an open ISO standard having both proprietary and open source implementations (e.g., Microsoft .NET and Rotor/SSCLI, Novell Mono) available on at least five major operating systems (BSD, Linux, MacOS X, Solaris, and Windows). Given that C++/CLI exists to support C++ for CLI development, C++/CLI should be available where CLI is, and third parties should be able to freely implement it for those other platforms if they feel that it is useful to do so. (See also Figure 15, taken from [N1557] slides 59-62.)

Interoperability is also a priority for C++/CLI. CLI is designed to be an environment where completely different languages can interoperate, and so two C++/CLI implementations would more or less automatically be compatible in the pure-CLI subset. But it would be a shame if it was not guaranteed that CLI libraries produced by two different C++/CLI compilers will be fully interoperable also in their uses of C++/CLI-specific extensions (e.g., ensuring that **const** parameters on CLI type member functions are

expressed and recognized the same way in metadata). So a key reason to standardize C++/CLI was to standardize also runtime compatibility between C++/CLI implementations for C++/CLI-specific features and extensions. (Note: C++/CLI does not try to specify binary compatibility between the pure C++ parts of the two implementations, which is left unspecified by the C++ standard.)

Did the C++/CLI committee (TG5) closely coordinate C++/CLI's development with the ISO C++ committee (WG21)?

Yes. TG5's members are a subset of WG21, including many of WG21' s best-regarded and trusted experts, and TG5 has always asked WG21 for direction and worked closely with ISO C++. In addition, because this work was related to the existing ISO C++ standard, Ecma granted full access to ISO members to participate in TG5 meetings and email technical discussions, and much input from ISO C++ participants is already reflected in C++/CLI. All C++/CLI documents have been in the WG21 mailings and available to WG21 national body members. TG5 actively worked to eliminate conflicts between C++/CLI and both ISO C++ and future C++0x evolution by discussing all overlapping feature proposals at WG21's meetings to get WG21's input and direction, and TG5 followed the WG21 evolution working group's direction on how best to stay out of the way of C++0x's own contemplated extensions to C++.

Is C++/CLI a "fork" of C++?

No. C++/CLI has taken great pains to be fully compatible with C++, including with future C++0x evolution, and the participants are determined that C++/CLI will continue to change as needed to continue to faithfully and compatibly track C++ as ISO C++ itself changes in the future. (See previous question.)

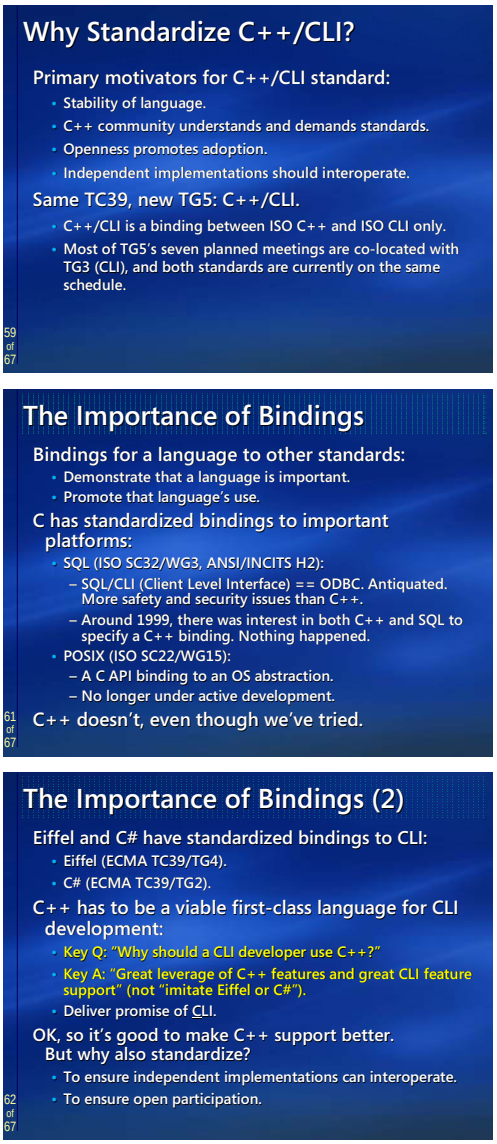


Figure 15: A rationale for standardizing C++/CLI (from [N1557])

Is C++/CLI now a standard?

Yes. C++/CLI is Ecma standard Ecma-372, adopted in December 2005. C++/CLI was then submitted by Ecma to ISO as a proposed ISO standard.

What's an ISO fast-track submission, and why is C++/CLI pursuing that?

The fast-track process was created by ISO to facilitate ISO adoption of open standards produced by other accredited international standards organizations that follow procedures approved of by ISO (i.e., are recognized to be legitimately open, produce consensus standards of high quality, etc.); Ecma is one of those accredited organizations. The first CLI standard was produced by Ecma and then submitted to ISO for fast-track processing, and as of this writing the second edition of the CLI standard is following the same process. It was natural that C++/CLI follow the same path as CLI so as to stay close to where CLI was

being maintained and under active revision, so as to be able to coordinate with it and influence it as needed, and for C++/CLI to be available in the same places as CLI.

But that doesn't mean that submitting C++/CLI as an ISO fast-track proposal is the best choice or the only choice, but just the default choice. Since 2003 TG5 has communicated to WG21 that this was the default plan along with regular progress updates, and TG5 has actively sought WG21's consensus direction on what WG21 wanted to see happen with C++/CLI, up to and including whether WG21 wanted it to take some other route, ranging from not entering ISO at all to having WG21 actually take over ownership of the C++/CLI work to do with as it will, and TG5 has left those questions to WG21's decision. WG21 hasn't yet agreed on a consensus, but TG5 has always asked and is continuing to ask for WG21 direction on this.

Why are these extensions called "C++/CLI"?

Because the extensions are designed to harmonize the features of the ISO C++ and ISO CLI standards. C++/CLI seemed to be the most obvious name, connotes putting C++ first before CLI, and deliberately avoids the form "*adjective* C++."

> *I picked C++ because it was short, had nice interpretations, and wasn't of the form "adjective C."*
>
> *— B. Stroustrup (D&E, p. 64)*

Until the middle of 2003, the internal name being used was "MC^2" which wanted to connote "M[anaged] C[++]" and the ideas of "squared," "version 2," and "with a ^." I thought that was too cute. More importantly, one problem Microsoft had already encountered with the name Managed Extensions was that people would too often informally shorten the name to "Managed C++," and then a vocal minority of people read various incorrect implications into that short form (e.g., that the extensions were intended to create a separate or successor language, or that Microsoft was trying to "manage" C++). So I tried to get away from any name that had a connection with that shortening (including anything with the word "managed" or the letters "MC"), or any new name that itself would be likely to be shortened to something like "*adjective* C++," because the community had objected to that. In 2002 and 2003 I asked a number of WG21 experts about this and no one could come up an alternative they felt would be significantly better, so I suggested C++/CLI as the working name because it was short and clear, and no one raised any concern about it until after the Ecma standard was ratified in December 2005.

Some people think "C++/CLI" sounds too much like C++. Couldn't it be called something else?

Sure. A request to change the name was raised in January 2006, and TG5 is now consulting WG21 for direction at the next WG21 meeting in April 2006.

Why does C++/CLI seem to add a separate language feature for each feature of CLI subjectively considered important by the C++/CLI design team?

Separate features were added only where reusing another C++ or C++/CLI feature couldn't be made to work, and the decision of which features to support was mostly not subjective.

An essential design goal of C++/CLI was to give C++ programmers first-class support for CLI programming without requiring them to switch to a different language (see §1); otherwise, there would be no point to the exercise. The CLI's Common Language Specification (CLS) sets out a standard list that is the consensus of independent experts representing a dozen and more different programming languages, including C++, of the minimum subset of CLI features that languages should be expected to support as a CLS consumer (able to use existing CLI libraries) and/or CLS extender (able to author new CLI libraries).

Most of the extensions in C++/CLI are required just to be a conforming CLS consumer and extender. Some other features were included in order to reach the additional goal of preserving C++'s position as a systems programming language, by leaving no room for a CLI language lower than C++.

Does CLI just expect languages to conform to its rules, or did CLI seriously make design decisions to consider C++?

CLI has specifically made design decisions to be considerate of C++, including in the CLI type system.

One example is the way CLI treats members name lookup: CLI supports both **HideByName** and **HideBySig** member function lookup, where a member function **f(int)** in a derived class either hides all base class member functions with the same name **f** (as in C++) or hides only base class member functions with the same signature **f(int)** (as in many other languages). This was specifically added to enable latitude for C++'s hide-by-name lookup rules.

A related example is the way CLI treats overriding: CLI permits overriding a private virtual function specifically because C++ allows those semantics. (Many other languages treat accessibility and overridability the same way, so that a virtual function must be accessible in order to be overridable.)

A third example is CLI operators: CLI made a good faith effort to support all the C++ operators, including overloaded assignment operators. (Unfortunately, CLI made the mistake of having those operators use pass-by-value for both parameters and return values, which does not support chainable assignment operators.)

Isn't it true that most of the facilities added could and should have been added as libraries? Did the C++/CLI design team just feel that only built-in language facilities would be acceptable to programmers?

No. The two main issues motivating language extensions are that CLI features behave differently enough from C++ features that the differences can't be correctly papered over (as the Managed Extensions tried valiantly to do), and/or they require compiler support to recognize the features and do correct code generation. Normally, programming language design is about finding the right abstractions to surface the language's chosen semantics. In this case, CLI's abstractions already existed with prescribed semantics and metadata representations, which made the design work more about finding the cleanest and most seamless way of surfacing the abstractions.

Why is feature X necessary (or, a language instead of a library feature; or, spelled that way; or, …)?

See the body of this paper for a rationale of representative C++/CLI features, such as **ref class**, **property**, **^**, and **generic**.

Is portable C++ code important? Do you recommend that application developers should wrap CLI-specific code modules that expose portable C++ interfaces to the rest of the application?

Yes. I strongly recommend this, and I am happy that it is the way the programmers I have seen are actually using C++/CLI in practice (see Appendix for an example). It is always a good practice to contain system-specific code (and even code specific to a third-party library vendor) in system-specific modules that are wrapped with portable C++ interfaces. I always did this when I developed commercial third-party software products; it insulates the application from changes in the underlying system or libraries and lets the developer switch implementations more readily. Note that a technology like C++/CLI is still necessary internally to implement those wrappers because you can't use CLI libraries without compiler support, but it's a good thing to use those extensions only in the places that need to use CLI.

Doesn't C++/CLI add considerable confusion by changing some of C++ rules for CLI classes? For example, with CLI types, inheritance is public instead of private by default, there are new conversion functions, and virtual function dispatch during construction and destruction works differently.

No, and this question reflects a basic misunderstanding. C++/CLI doesn't change C++ rules; it faithfully preserves correct C++ rules for C++ types, and correct CLI rules where they are inherent for CLI types. The features mentioned above are aspects of how CLI types inherently do behave that can't be fully hidden behind the programming model.

The lack of direct expression of such essential behaviors of CLI types was one of the reasons C++ was unusable and irrelevant on the ISO CLI platform before C++/CLI. Note that the Managed Extensions designers tried to paper over too many of such differences, and the result based on experience in the field with two shipped releases is that this failed to work for programmers and caused great confusion.

Note that C++/CLI can and does nonintrusively *add* C++ features onto CLI types, using the CLI's facilities that support such extended semantics. For example, C++/CLI enables CLI reference types to be templates, have member function templates, have copy constructors and copy assignment operators, and have **const** data members and function parameters. Also, wherever possible C++ rules are preferred even for CLI types. For example, when searching through base classes to look up the name of a member function, C++/CLI first applies only C++ hide-by-name rules; only if that fails to find a candidate, so that the code would be an error in the pure C++ rules, does it fall back to repeat the lookup applying hide-by-signature lookup rules which makes more function names visible.

What a language usually cannot do is *change* existing CLI behaviors like the above that are inherent in the CLI environment and object model. The C++/CLI designers explored many alternatives to try to force at least some C++ semantics on CLI types authored in C++/CLI (e.g., to force virtual function calls during construction to behave according to C++ rules), and this approach has important problems including:

- **It would create confusion, not reduce it.** For example, instead of just C++ types and CLI types, programs would have to know that there would now be C++ types, CLI types that behave like CLI types (e.g., in all existing CLI libraries), and CLI types that partly behave like C++ types (and which would not work well or at all when used by other CLI languages). I do not consider such extra complexity to be an improvement.
- **It is not always technically possible.** For example, for CLI value types it is not possible to support user-defined copy construction with correct semantics, because the CLI runtime creates bitwise copies at times where the compiler cannot insert a function call to invoke the copy constructor. (On the other hand, supporting copy constructors on CLI reference types works fine, as noted above.)

So C++/CLI avoids confusion by surfacing both C++ and CLI semantics correctly — correct C++ rules for C++ types, and correct CLI rules where they are inherent for CLI types. That's not a bug, it's an essential feature. And C++CLI surfaces the CLI semantics with the minimum possible intrusiveness: For example, to denote CLI type semantics, the programmer authoring a new CLI type writes one word (e.g., **ref** or **value**, once on the declaration of the class only) to tag that "this set of rules applies," after which the code that uses the type just instantiates and uses objects as usual without any further syntax extensions. One word per type for the above-cited rules is about as low-impact as it's possible to get.

At the time this project was launched in 2003, participants described it as an attempt to develop a "binding" of C++ to CLI, and a minimal (if still substantial) set of extensions to support that environment. But C++/CLI has effectively evolved into a different language. Hasn't it?

First, this question incorrectly implies that what was presented in 2003 is not what was delivered. Anyone can compare for themselves — see:

- [N1557], the 2003 overview presentation to the ISO C++ committee, available on the ISO C++ committee website. That presentation covered the same ground as this paper, and this paper frequently reproduces its slides for reference and comparison.
- [C++/CLI], the current Ecma standard.

In particular, the [N1557] presentation included:

- Slides 5-6: The same goals covered in this paper. (See §1.)
- Slides 7-10: The same technical constraints covered in this paper, including the need for language extensions because CLI types behave differently (specifically mentioning the issues of deep virtual calls in constructors, and the dual nature of value types), as well as the use of contextual keywords. (See §1.2 and §3.1.)
- Slide 14: The point of why defaults should be different on CLI class types. (See §3.1.5.)
- Slide 17: The contemplated future unification of allowing overloaded and templated property setter functions. (See §3.2.3.)
- Slide 30: The ability to write agnostic templates that can work seamlessly on both C++ and CLI types and features. (See §3.3.3.)
- Slides 30-38: Why C++ destructors are a huge strength and why they make CLI types easier to use, even ones authored in other languages. (See §3.1.3.)[18]
- Slides 41-43: Why generics are distinct from templates, but how they can be integrated (e.g., generics can match template template parameters, and a mention of the STL/CLI project). (See §3.4.)
- Slides 51-53: The contemplated future unification supporting cross-inheritance between C++ and CLI types (aka "mixed types"). (See §3.1.3.)
- Slides 55-57: Pure extensions to ISO C++, including even macro compatibility, via contextual keywords with a note that these are a burden on compiler writers to implement but were chosen because the designers viewed ISO C++ compatibility as more important. (See §1.1 and §1.2.)
- Slides 59-64: Motivations and the contemplated timeline and track for C++/CLI standards work (Ecma, then ISO fast-track submission). The presentation and further discussion that week in 2003 included explicit requests for direction on whether WG21 felt this was the right plan. Individuals raised various concerns, but WG21 did not decide to direct TG5 to pursue a different course and TG5 and WG21 planned close technical liaison to ensure ISO C++ compatibility (which both followed through on).

The main differences I can see between what was presented and the way C++/CLI ended up are:

- It took an extra year to do the technical work than the original timeline estimated.
- C++/CLI did manage to avoid making **generic** a reserved word after all, further improving ISO C++ compatibility.
- C++/CLI ended up not supporting copy constructors and copy assignment operators on value types. It turned out that the assumption that CLI value type instances are bitwise copyable is inherent in CLI, and CLI has a few places where the runtime has latitude to make bitwise copies of value type instances but where the C++/CLI compiler cannot inject code to make a function call and so guarantee that the copy constructor or copy assignment operator will be called. Since the compiler

[18] See also my many blog entries about this, as well as my 2004 OOPSLA talk on this topic which is currently available as streamed video via a link on my home page www.gotw.ca. Disclaimer: I have no control over the technologies used to create the streamed version of this video, and it doesn't work on all browsers and players; I'm just linking to it so you can view it if you have compatible software.

can't guarantee that these functions will be called, it would be wrong to let programmers write them and have them only mostly work.

- C++/CLI ended up not including the unification of permitting CLI types to be allocated on the C++ heap via **new**, and C++ types to be allocated on the CLI heap via **gcnew** (including the related feature of allowing finalizers on C++ types), though it deliberately left room for that unification as an intended future extension.

As for whether this set of extensions amounts to a different superset language, a compatible dialect, and/or a binding: I think you can find reasonable people who view it any of those ways. Whichever noun you prefer, it is the most compatible set of extensions I know of to any programming language, standard or otherwise, including for nearly all macro cases — which are notoriously next to impossible to support in a compatible way. Achieving that meant putting requirements in C++/CLI that placed a greater burden on compiler writers in favor of preserving strong ISO C++ compatibility and avoiding conflict with C++0x evolution, and improving programmer usability so that programmers would use the feature instead of switch to another language (unlike with the Managed Extensions). These choices reflect the designers' great respect for ISO C++.

5 Glossary

assembly — A CLI library or executable, containing the metadata describing the library's contents.

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

C++0x — The next revision of ISO C++ currently under development in ISO WG21 (ISO C++ standards committee). (See also [C++], the current standard.)

C++/CLI — A set of extensions that harmonize ISO C++ and ISO CLI programming, so that C++ programmers are able to use CLI libraries and create new ones. Supersedes the Managed Extensions. (See also "Managed Extensions.")

CLI — Common Language Infrastructure (see [CLI]). The Ecma and ISO standardized subset of the .NET runtime and base class library. There are at least two commercial implementations of CLI (Microsoft .NET and Novell Mono) as well many research and other implementations.

CLI library — See Assembly.

CLS — The CLI Common Language Specification defined in [CLI §7], which defines minimum requirements for CLI-compatible languages including requirements to be a CLS extender and a CLS consumer. "The CLS is designed to be large enough that it is properly expressive yet small enough that all languages can reasonably accommodate it." [CLI §7.2]

CLS framework — "A library consisting of CLS-compliant code." [CLI §7.2.1]

CLS consumer — "A CLS consumer is a language or tool that is designed to allow access to all of the features supplied by CLS-compliant frameworks, but not necessarily be able to produce them. The following is a partial list of things CLS consumer tools are expected to be able to do: […] Create an instance of any CLS-compliant type" and otherwise support consuming libraries that use major CLI features. [CLI §7.2.2]

CLS extender — "A CLS extender is a language or tool that is designed to allow programmers to both use and extend CLS-compliant frameworks. CLS extenders support a superset of the behavior supported by a CLS consumer (i.e., everything that applies to a CLS consumer also applies to CLS extenders). In addition to the requirements of a consumer, extenders are expected to be able to: Define new CLS-compliant types that extend any (non-sealed) CLS-compliant base class …" and support authoring other CLI features. [CLI §7.2.3]

CLI reference type — A normal CLI type that can be physically allocated only on the CLI heap. Denoted using **ref class** in C++/CLI.

CLI value type — An efficient CLI type that models a value and therefore is safe for bitwise copying. Every value type actually has two forms, which are semantically different and so can be considered distinct, though implicitly related, types: 1. The usual unboxed form, which is always instantiated directly on the stack or as a directly embedded member of another object, and does not have a vtable. 2. The boxed form, which is a standalone heap object with a vtable (this indirection is denoted in C++/CLI using a handle to a value type). (See "boxing.") Denoted using **value class** in C++/CLI.

handle — A C++/CLI handle is called an "object reference" in the CLI specification. For a CLI class **T**, a **T^** is a handle to an object of type **T** on the CLI heap. A handle tracks, is rebindable, and can point to a whole object only. (See also "tracking.")

heap, CLI — The storage area (accessed by **gcnew**) that is under the control of the garbage collector of the virtual machine as specified in the CLI.

heap, C++ — The storage area (accessed by **new**) as defined in the C++ Standard [C++ §18.4].

ILASM — CLI instruction language assembler.

Managed Extensions — A precursor to C++/CLI, the first attempt to define a set of extensions to C++ to enable CLI programming. The Managed Extensions specified fewer extensions to C++ than does C++/CLI, and failed to gain acceptance among programmers in large part for that reason; its frugality often obscured essential differences and was confusing to use. This set of extensions is also commonly called "Managed C++." It shipped in Visual C++ 2002 and 2003, and is supported but deprecated in Visual C++ 2005.

metadata — Data that describes and references the types defined by the CLI 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 CLI virtual machine.

pinning — Preventing an object on the CLI heap from moving during garbage collection, so that for the duration of the pin it is safe to use a normal C++ pointer to the object. Holding a pin interferes with CLI heap garbage collection, and so pins are intended to be used sparingly and to have short lifetimes.

reference type — See CLI reference type.

value type — See CLI value type.

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 C++ heap, which never move). Tracking is maintained by the virtual machine during garbage collection. Tracking is an inherent property of C++/CLI handles.

unboxing — An explicit conversion from CLI type **System::Object^** to any CLI value class type, from CLI type **System::ValueType^** to any CLI value class type, from **V^** (the boxed form of a CLI value class type) to **V** (the normal unboxed form of the CLI value class type), or from any CLI interface class type handle to any CLI value class type that implements that CLI interface class.

virtual machine — Called the Virtual Execution System (VES) in the CLI standard. 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 CLI code and data, using the metadata to connect separately generated modules together at runtime. For example, given an address inside the code for a function, it must be able to locate the metadata describing that function. It must also be able to walk the stack, handle exceptions, and store and retrieve security information.

6 References

[C++] ISO/IEC International Standard 14882:2003, *Programming Languages — C++* (2nd ed., April 2003).

[C++PL3e] B. Stroustrup. *The C++ Programming Language, 3rd edition* (Addison-Wesley, 1997).

[C++/CLI] Standard Ecma-372, *C++/CLI Language Specification* (1st ed., December 2005).

[CLI] Standard Ecma-335, *Common Language Infrastructure* (3rd ed., June 2005).

[CLI03] ISO/IEC International Standard 23271:2003, *Common Language Infrastructure (CLI)* (1st ed., April 2003). [CLI] is an updated edition that has been approved by Ecma and is now under ISO consideration.

[D&E] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).

[N1557] H. Sutter. "C++/CLI Overview" (presentation to ISO C++ standards committee, October 2003).

Appendix: A Typical Public User Comment

> *C++ was designed primarily so that the author and his friends would not have to program in assembler, C, or various modern high-level languages. Its main purpose is to make writing good programs easier and more pleasant for the individual programmer.*
>
> *— B. Stroustrup (D&E, p. 106)*

The following unsolicited comment is representative of people who are actually using a C++/CLI compiler (currently Visual C++ 2005), and is typical of how C++/CLI is often used in practice in the field.[19]

Color emphasis is added. Ellipses are original.

In comp.lang.c++.moderated on 29 January 2006, Andrew Browne wrote:

I have made some (so far) limited use of C++/CLI in developing a web front end to some core application code written in standard C++. This has been my first experience of writing web front end software and I have greatly appreciated the fact that I can write this essentially in the language I know and love (rather than, say, JavaScript) and that calls to my standard C++ core application code (which I haven't needed to change) are natural and seamless. Furthermore in this development I have been able to make seamless use of third party components which happen to have been written in different programming languages (C# and VB.NET.) I have also found that I have needed to make only very limited use of the new keywords and syntax specific to C++/CLI to do all this. Of course I would prefer to be able to do it all entirely in standard C++ without any new keywords or syntax and if there is a current technology which would let me do all this in standard C++ alone then I would be very interested to know about it…

For me, CLI (or .NET) seems to offer the possibility of developing web applications and web services in a way which is not too alien to the sort of programming I am used to, and in which I can continue to use my favourite programming language pretty much the whole time. I will continue to develop my core application code in standard, portable C++, and I see no good reason to do otherwise, but C++/CLI seems to me, so far, to be quite a promising tool for making use of that application code in a web environment, especially if C++/CLI compilers become available for Mono or other non-Microsoft CLI implementations as I hope they will.

I also hope that, in future, C++/CLI development will move yet closer to standard C++ rather than further away…

Andrew

[19] See also my blog article "How much C++/CLI syntax do you need to use all of the .NET Frameworks?" (December 15, 2003).