C++/CLI Overview

Herb Sutter

Architect
Microsoft Visual C++

Quake II Takeaways

960x720 + software-rendered on 1.2GHz PIII-M + Fx 1.1.4322

1. It's easy to run existing C/C++ code on CLI: 100% JITted (IL) code; still native data.
   - Just rebuild with /clr.
   - 1 day to port the entire Quake 2 source base. (Nearly all of the effort was to translate from C to C++, and had nothing to do with our compiler or the CLI platform.)
2. It's not hard to extend existing code with CLI types.
   - 2 days to implement the radar extension using Fx (gradient brushes, window transparency/opacity, Matrix.RotateAt).
3. It needs to be still easier, more natural, and "first-class" to use C++ on the CLI.

2 of 67

Some Definitions

ECMA: European Computer Manufacturers' Association.

- Accredited ISO fast-track submitter.
- TC39: Programming language technical committee. ("SC22")

CLI: Common Language Infrastructure.

- The ECMA- and ISO-standardized part of the CLR (Common Language Runtime, virtual machine with garbage collection), Base Class Library (BCL), and Frameworks (Fx).
  - ECMA TC39/TG3: TG maintaining the CLI standard.

IL: Intermediate language.

- The instruction set of the virtual machine. IL has OO concepts baked in: Base classes, virtual function dispatch, casting, etc.

JIT: Just-in-time compilation to native machine code.

Verifiability: Code that can be proven "correct."

- Examples: No type errors, no array overruns.

3 of 67

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unified pointer and storage system (stack, native heap, gc heap).
   - Deterministic cleanup: Destruction/Dispose, finalization.
   - Generics × templates, STL on CLI.
   - Unified type system, mixing native/CLI, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

4 of 67

Rationale

C++: First-class CLI development language.

- Remove "Why Can't I" usability and migration barriers: Port and extend existing programs even more seamlessly.
- Key Qs: "Why should a CLI developer use C++?"
  "Is C++ relevant in VM environments with GC?"
- Deliver promise of CLI.

"Managed C++" insufficient: Grafting vs. integration.

- Great for basic interop, migrating existing code to CLI.
- Poor exposure of CLI features (e.g., __property). Poor integration of C++ and CLI features (e.g., no templates of CLI types). Hard to write pure (verifiable, secure) CLI apps.
- Ugly and nonintuitive syntax, uneven and contorted semantics. Failed to achieve a natural, organic, "everything in its place" surfacing of features.
- Low adoption. And those who do adopt still need to hand-wire way too much.

5 of 67

Major Goals

Feature coverage:

- Provide organic support for CLI features/idioms.
- Make sure they have a first-class feel.
  - Example: Verifiability at first try in this complete program:

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

- Leave no room for a language lower than C++ (incl. IL).

C++ × CLI: Why a CLI programmer should use C++.

- "Bring C++ to CLI": Support C++'s powerful features also for CLI types (e.g., deterministic cleanup, templates).
- "Bring CLI to C++": Support the CLI's powerful features also for native types (e.g., verifiability, garbage collection).

6 of 67

Major Constraints

A binding: Not a commentary or an evolution.

- No room for “while we’re at it…” thinking.

Conformance: Prefer pure conforming extensions.

- Nearly always possible, if you bend over backward far enough. Sometimes there’s pain, though.
  - Attempt #1: __ugly_keywords. Users screamed and fled.
  - Now: Keywords that are not reserved words, via various flavors of contextual keywords.

Usability:

- More elegant syntax, organic extensions to ISO C++.
- Principle of least surprise. Keep skill/knowledge transferable.
- Enable quality diagnostics when programmers err.

7 of 67

Corollary: Basic Hard Call #1

“Pure extension” vs. “first-class feel”?

- Reserved keywords give a better programmer experience and first-class feel. But they’re not pure extensions any more.

Our evaluation: Both purity and naturalness are essential.

- So we have to work harder at design and implementation.
- Good news for conformance: Currently down to only three reserved words (generic, gcnew, nullptr).
- Good news for the user: There are other keywords – they’re just not reserved words. This retains a first-class experience.
- Hard work for language designers and compiler writers: Extra effort via extra parsing work and a lex hack.

8 of 67

Corollary: Basic Hard Call #2

"Don't comment" vs. "orthogonality"?

- Orthogonal features are good: They make learning easier and make programmers more productive.
- They can look like commentary even though they're not.

Our evaluation: Orthogonality is essential.

- Inconsistency, unevenness, and special cases were a huge source of complaints about "Managed C++":
  - T* meant 3 different & incompatible things, depending on T.
  - Gc and properties for CLI types, but not native ones.
  - Auto destruction and templates for native types, not CLI ones.
- Insist on supporting features uniformly: "This is how you do it" for any type T.
  - The easy sell: "Great, C++ is showing 'em how to do it right!"
  - The corollary: "Hey, they're invading our C++!"
  - Warn by default when extensions are used on native types.

9 of 67

Why a Language-Level Binding

Reference types:

- Objects can physically exist only on the gc heap.
- Deep virtual calls in constructors.

Value types:

- Cheap to copy, value semantics.
- Objects physically on stack, gc heap, & some on native heap.
  - Gc heap: "Boxed," full-fledged polymorphic objects (e.g., Int32 derives from System::Object, implements interfaces).
  - Otherwise: Laid out physically in place (not polymorphic).

Interfaces:

- Abstract. Only pure virtual functions, no implementations.
- A lot like normal C++ abstract virtual base classes.

10 of 67

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unified pointer and storage system (stack, native heap, gc heap).
   - Deterministic cleanup: Destruction/Dispose, finalization.
   - Generics × templates, STL on CLI.
   - Unified type system, mixing native/CLI, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

11 of 67

adjective class C;

12 of 67

Basic Class Declaration Syntax

Type are declared "*adjective* class":

```
          class N{ /*...*/ };    // native
      ref class R { /*...*/ };   // CLI reference type
    value class V { /*...*/ };   // CLI value type
interface class I  { /*...*/ };  // CLI interface type
     enum class E { /*...*/ };   // CLI enumeration type
```

- C++ & CLI fundamental types are mapped to each other (e.g., int and System::Int32 are the same type).

Any type can:

- Have a destructor ~T(), and/or finalizer !T().
- Have a copy constructor, and/or copy assignment operator:
  - Value classes always have them. Native classes have them by default. Ref classes do not have them by default.
- Be templated, or be used to instantiate a template.
- (More on each of these later on.)

13 of 67

Class Declaration Extensions

Abstract and sealed:

```
ref class A abstract { }; // abstract even w/o pure virtuals
ref class B sealed : A { };      // no further derivation is allowed
ref class C : B { };             // error, B is sealed
```

Things that are required anyway are implicit:

- Inheritance from ref classes and interfaces is implicitly public. (Anything else would be an error, so why make the programmer write out something that is redundant?)

```
ref class B sealed : A { };           // A is a public base class
ref class B sealed : public A { };    // legal, but redundant
```

- Interfaces are implicitly abstract, and an interface's members are implicitly virtual. (Ditto the above.)

```
interface class I { int f(); };   // f is pure virtual
```

CLI enumerations:

- Scoped. Can specify underlying type. No implicit conversion to int.

14 of 67

Properties

Basic syntax:

```
ref class R {
  int mySize;
public:
  property int Size {
    int get()            { return mySize; }
    void set( int val )  { mySize = val; }
  }
};
R r;
r.Size = 42;             // use like a field; calls r.Size::set(42)
```

Trivial properties:

```
ref class R {
public:
  property int Size;     // compiler-generated
};                       // get, set, and backing store
```

15 of 67

Indexed Properties

Indexed syntax:

```
ref class R { // ...
  map<String^,int>* m;
public:
  property int Lookup[ String^ s ] {
    int get()          { return (*m)[s]; }
  protected:
    void set( int );   // defined out of line below
  }
  property String^ default[ int i ] { /*...*/ }
};
void R::Lookup[ String^ s ]::set( int v ) { (*m)[s] = v; }
```

Call point:

```
R r;
r.Lookup["Adams"] = 42;    // r.Lookup["Adams"].set(42)
String^ s = r[42];         // r.default[42].get()
```

16 of 67

Contemplated Orcas Extensions

Overloaded and templated setters:

```
ref class R {
public:
  property Foo Bar {
    Foo get();
    void set( Foo );
    void set( int );          // overloaded function
    template<class T>         // overloaded function template
    void set( T );
  }
};
```

17 of 67

Delegates and Events

A trivial event:

```
delegate void D( int );
ref class R {
public:
  event D^ e;    // trivial event; compiler-generated members
  void f() { e( 42 ); }     // invoke it
};

R r;
r.e += gcnew D( this, &SomeMethod );
r.e += gcnew D( SomeFreeFunction );
r.f();
```

Or you can write add/remove/raise yourself.

- Contemplated for Orcas: Overloaded/templated raise.

18 of 67

Virtual Functions and Overriding

Explicit, multiple, and renamed overriding:

```
interface class I1 { int f();   int h();                };
interface class I2 { int f();          int i();         };
interface class I3 {                   int i();  int j(); };

ref struct R : I1, I2, I3 {
  virtual int e() override;     // error, there is no virtual e()
  virtual int f() new;          // new slot, doesn't override any f
  virtual int f() sealed;       // overrides & seals I1::f and I2::f
  virtual int g() abstract;     // same as "= 0" (for symmetry
                                   with class declarations)

  virtual int x() = I1::h;      // overrides I1::h
  virtual int y() = I2::i;      // overrides I2::i
  virtual int z() = j, I3::i;   // overrides I3::i and I3::j
};
```

- See also:
  Stroustrup & O'Riordan's 1990/1 paper with similar syntax.
  Gutson's paper N1494 in the current mailing.

19 of 67

CLI Enums

Three differences:

- Scoped.
- No implicit conversion to underlying type.
- Can specify underlying type (defaults to int).

```
enum class E1 { Red, Green, Blue };
enum class E2 : long { Red, Skelton };
E1 e1 = E1::Red;          // ok
E2 e2 = E2::Red;          // ok
e1 = e2;                  // error
int i1 = (int)Red;        // error
int i2 = E1::Red;         // error, no implicit conversion
int i3 = (int)E1::Red;    // ok
```

- See also Miller's paper N1513 in the current mailing.

20 of 67

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
    - Unified pointer and storage system (stack, native heap, gc heap).
    - Deterministic cleanup: Destruction/Dispose, finalization.
    - Generics × templates, STL on CLI.
    - Unified type system, mixing native/CLI, other features.
4. C++/CLI Standardization
    - Venue, players, timelines, how to participate.

21 of 67

% is to ^

as

& is to *

22 of 67

Unified Storage/Pointer Model

Semantically, a C++ program can create object of any type T in any storage location:

- On the native heap (lvalue): `T* t1 = new T;`
  - As usual, pointers (*) are stable, even during GC.
  - As usual, failure to explicitly call delete will leak.
- On the gc heap (gc-lvalue): `T^ t2 = gcnew T;`
  - Handles (^) are object references (to whole objects).
  - Calling delete is optional: "Destroy now, or finalize later."
- On the stack (lvalue), or as a class member: `T t3;`
  - Q: Why would you? A: Next section: Deterministic destruction/dispose is automatic and implicit, hooked to stack unwinding or to the enclosing object's lifetime.

Physically, an object may exist elsewhere.

23 of 67

Pointers

Native pointers (*) and handles (^):

- ^ is like *. Differences: ^ points to a whole object on the gc heap (gc-lvalue), can't be ordered, and can't be cast to/from void* or an integral type. (There is no void^.)

```
Widget* s1 = new Widget;      // point to native heap
Widget^ s2 = gcnew Widget;    // point to gc heap
s1->Length();                 // use -> for member access
s2->Length();
(*s1).Length();               // use * to dereference
(*s2).Length();
```

Use RAII pin_ptr to get a * into the gc heap:

```
R^ r = gcnew R;
int* p1 = &r->v;              // error, v is a gc-lvalue
pin_ptr<int> p2 = &r->v;      // ok
CallSomeAPI( p2 );            // safe call, CallSomeAPI( int* )
```

24 of 67

References and Unary &/%

Native (&) and tracking (%) references:

- % is like &. Differences: % can refer into any memory area incl. the gc heap (binds to any lvalue or gc-lvalue). For now, a % can only exist on the stack.

```
String& s3 = *s1;                  // bind
String% s4 = *s2;                  // bind & track
s3.Length();                       // reference syntax with .
s4.Length();
void swap( Object^% o1, Object^% o2 )      // C# "ref"
  { Object^ tmp = o1; o1 = o2; o2 = tmp; }
```

Unary & and % for "address of":

- &myobj → MyType*
  (or interior_ptr<MyType>, for a gc-lvalue).
- %myobj → MyType^.

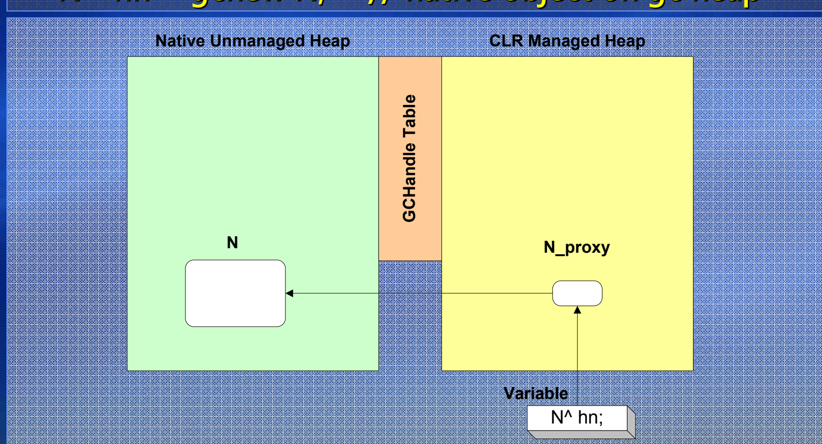
25 of 67

Native on the GC Heap

Create a proxy for native object on gc heap.

- The proxy's finalizer will call the destructor if needed.

```
N^ hn = gcnew N;     // native object on gc heap
```



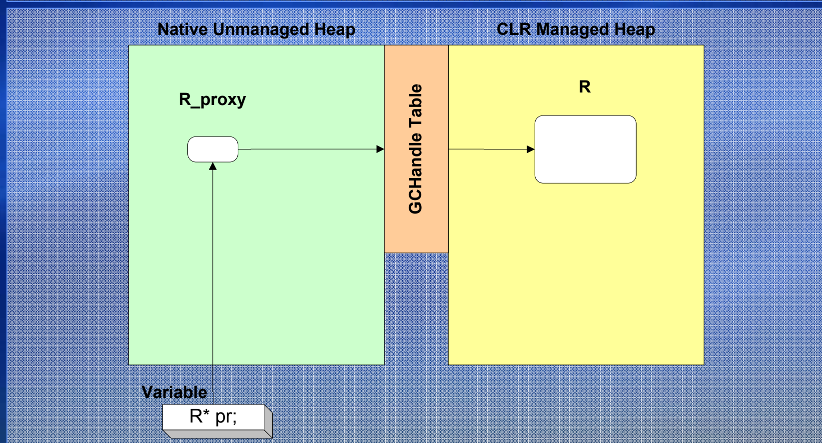
26 of 67

Ref Class on Native Heap

Already implemented as gcroot template.

- No finalizer will ever run. Example:

```
R* pr = new R;        // ref object on native heap
```

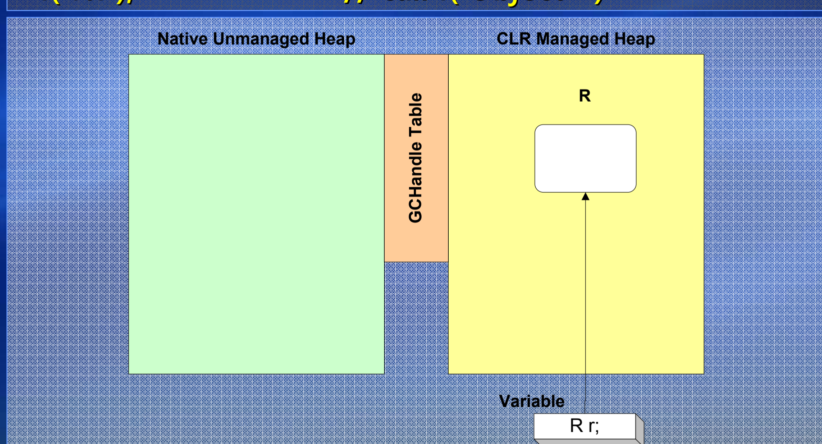


27 of 67

Ref Class on the Stack

The type of “%R” is R^.

```
R r;                  // ref object on stack
f( %r );              // call f( Object^ )
```



28 of 67

Boxing (Value Types)

Boxing is implicit and strongly typed:

```
int^ i = 42;        // strongly typed boxed value
Object^ o = i;      // usual derived-to-base conversions ok
Console::WriteLine( "Two numbers: {0} {1}", i, 101 );
```

- i is emitted with type Object + attribute marking it as int. WriteLine chooses the Object overload as expected.
- Boxing invokes the copy constructor.

Unboxing is explicit:

- Dereferencing a V^ indicates the value inside the box, and this syntax is also used for unboxing:

```
int k = *i;         // unboxing to take a copy
int% i2 = *i;       // refer into the box (no copy)
swap( *i, k );      // swap contents of box with stack variable
                    // (no copy, modifies the contents of box)
```

29 of 67

Aside: Agnostic templates

To demonstrate the unification, consider agnostic templates.

Example 1: Usual swap, with % instead of &.

```
template<class T>
void swap( T% t1, T% t2 )
  { T tmp( t1 ); t1 = t2; t2 = tmp; }
```

- Works for any copyable T:

```
Object ^o1, ^o2;          swap( o1, o2 );     // swap handles
int ^i1, ^i2;             swap( i1, i2 );     // swap handles
                          swap( *i1, *i2 );   // swap values
MessageQueue *q1, *q2;    swap( q1, q2 );     // swap pointers
                          swap( *q1, *q2 );   // swap values
ref class R { } r1, r2;   swap( r1, r2 );     // swap values*
value class V { } v1, v2; swap( v1, v2 );     // swap values
class Native { } n1, n2;  swap( n1, n2 );     // swap values*
```

* assuming copy construction/assignment are defined

30 of 67

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
    - Unified pointer and storage system (stack, native heap, gc heap).
    - Deterministic cleanup: Destruction/Dispose, finalization.
    - Generics × templates, STL on CLI.
    - Unified type system, mixing native/CLI, other features.
4. C++/CLI Standardization
    - Venue, players, timelines, how to participate.

31 of 67

T:: ~T()

and

T:: !T()

32 of 67

Cleanup in C++: Less Code, More Control

The CLI state of the art is great for memory.

It's not great for other resource types:

- Finalizers usually run too late (e.g., files, database connections, locks). Having lots of finalizers doesn't scale.
- The Dispose pattern (try-finally, or C# "using") tries to address this, but is fragile, error-prone, and requires the user to write more code.

Instead of writing try-finally or using blocks:

- Users can leverage a destructor. The C++ compiler generates all the Dispose code automatically, including chaining calls to Dispose. (There is no Dispose pattern.)
- Types authored in C++ are naturally usable in other languages, and vice versa.
- C++: Correctness by default, potential speedup by choice. (Other: Potential speedup by default, correctness by choice.)

33 of 67

Uniform Destruction/Finalization

Every type can have a destructor, ~T():

- Non-trivial destructor == IDispose. Implicitly run when:
  - A stack based object goes out of scope.
  - A class member's enclosing object is destroyed.
  - A delete is performed on a pointer or handle. Example:

```
Object^ o = f();
delete o;   // run destructor now, collect memory later
```

Every type can have a finalizer, !T():

- The finalizer is executed at the usual times and subject to the usual guarantees, if the destructor has not already run.
- Programs should (and do by default) use deterministic cleanup. This promotes a style that reduces finalization pressure.
- "Finalizers as a debugging technique": Placing assertions or log messages in finalizers to detect objects not destroyed.

34 of 67

Deterministic Cleanup in C++

C++ example:

```
void Transfer() {
  MessageQueue source( "server\\sourceQueue" );
  String^ qname = (String^)source.Receive().Body;
  MessageQueue dest1( "server\\" + qname ),
               dest2( "backup\\" + qname );
  Message^ message = source.Receive();
  dest1.Send( message );
  dest2.Send( message );
}
```

- On exit (return or exception) from Transfer, destructible/ disposable objects have Dispose implicitly called in reverse order of construction. Here: dest2, dest1, and source.
- No finalization.

35 of 67

Deterministic Cleanup in C#

Minimal C# equivalent:

```
void Transfer() {
 using( MessageQueue source
        = new MessageQueue( "server\\sourceQueue" ) ) {
  String qname = (String)source.Receive().Body;
  using( MessageQueue
    dest1 = new MessageQueue( "server\\" + qname ),
    dest2 = new MessageQueue( "backup\\" + qname ) ) {
   Message message = source.Receive();
   dest1.Send( message );
   dest2.Send( message );
  }
 }
}
```

36 of 67

Deterministic Cleanup in VB/Java

Alternative equivalent (in C# syntax):

```
void Transfer() {
  MessageQueue source = null, dest1 = null, dest2 = null;
  try {
    source = new MessageQueue( "server\\sourceQueue" );
    String qname = (String)source.Receive().Body;
    dest1 = new MessageQueue( "server\\" + qname );
    dest2 = new MessageQueue( "backup\\" + qname );
    Message message = source.Receive();
    dest1.Send( message );
    dest2.Send( message );
  }
  finally {
    if( dest2 != null ) { dest2.Dispose(); }
    if( dest1 != null ) { dest1.Dispose(); }
    if( source != null ) { source.Dispose(); }
  }
}
```

37 of 67

Deterministic Cleanup in C++ (2)

C++ example with polymorphism:

```
void Transfer() {
  auto_ptr<Object> source =
    new MessageQueue( "server\\sourceQueue" );
  // ...
}
```

38 of 67

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unified pointer and storage system (stack, native heap, gc heap).
   - Deterministic cleanup: Destruction/Dispose, finalization.
   - Generics × templates, STL on CLI.
   - Unified type system, mixing native/CLI, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

39 of 67

generic<typename T>

40 of 67

Generics × Templates

Both are supported, and can be used together.

Generics:

- Run-time, cross-language, and cross-assembly.
- Constraint based, less flexible than templates.
- Will eventually support many template features.

Templates:

- Compile-time, C++, and generally intra-assembly (a template and its specializations in one assembly will also be available to friend assemblies).
- Intra-assembly is not a high burden because you can expose templates through generic interfaces (e.g., expose a_container<T> via IList<T>).
- Supports specialization, unique power programming idioms (e.g., template metaprogramming, policy-based design, STL-style generic programming).

41 of 67

Generics

Generics are declared much like templates:

```
generic<typename T>
where T : IDisposable, IFoo
ref class GR { // ...
  void f() {
    T t;
    t.Foo();
  } // call t.~T() implicitly
};
```

- Constraints are inheritance-based.

Using generics and templates together works.

- Example: Generics can match template template params.

```
template< template<class> class V >  // a TTP
void f() { V<int> v; /*...use v...*/ }
f<GR>();                              // ok, matches TTP
```

42 of 67

STL on the CLI

C++ enables STL on CLI:

- Verifiable.
- Separation of collections and algorithms.

Interoperates with Frameworks library.

C++ "for_each" and C# "for each" both work:

```
stdcli::vector<String^> v;
for_each( v.begin(), v.end(), functor );
for_each( v.begin(), v.end(), _1 += "suffix" );   // C++
for_each( v.begin(), v.end(), cout << _1 );       // lambdas
g( %v );    // call g( IList<String^>^ )
for each (String^ s in v) Console::WriteLine( s );
```


Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unified pointer and storage system (stack, native heap, gc heap).
   - Deterministic cleanup: Destruction/Dispose, finalization.
   - Generics × templates, STL on CLI.
   - Unified type system, mixing native/CLI, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

ref class R : Native { };
class Native : R { };

45 of 67

CLI Types in the Native World

Basic interop example:

```
class Data {
  XmlDocument* xmlDoc;
public:
  void Load( std::string fileName ) {
    XmlTextReader^ reader = gcnew XmlTextReader(
                         marshal_as<String^>( fileName ) );
    xmlDoc = new XmlDocument( reader );
  }
};
```

46 of 67

CLI Types in the Native World (2)

Template<Ref> example:

```
template<class T>
void AFunctionTemplate( T ) { /*...*/ };
ref class Ref { /*...*/ };
Ref ref;
AFunctionTemplate( ref );               // ok
```

Of course, any type can be templated:

```
template<class T>
ref class ARefTemplate { /*...*/ };     // ok
```

47 of 67

Native Types in the CLI World

Basic interop example:

```
ref class MyControl : UserControl { //... // reference type
  std::vector<std::string>* words;        // use native type
public:
  void Add( String^ s ) { Add( marshal_as<std::string>(s)); }
  void Add( std::string s ) { words->push_back(s); }
};
```

Segueing to “futures”: Generic<Native> example.

```
generic<class T>
where T : I1
ref class SomeGeneric { /*...*/ };
class Native : I1 { /*...*/ };
SomeGeneric<Native> g;                  // ok
```

48 of 67

What Users Are Doing

Example 1: Quake 2 extension example (using v1 syntax):

```
private __gc class RadarForm
  : public System::Windows::Forms::Form
{
  std::vector<RadarItem>* m_items;
public:
  RadarForm() : m_items( new std::vector<RadarItem> )
    { /*...*/ };
  ~RadarForm() { delete items; }        // v1 finalizer syntax
  // ... etc.
};
```

- Their first attempt was without the * (i.e., they naturally tried make the vector a member), but that wasn't allowed.

49 of 67

What Users Are Doing (2)

Example 2: Faking up base classes (e.g., expose native types to a CLI world).

```
private __gc class C {        // can't inherit from Native, so...
  Native* n;
public:
  C() : n( new Native ) { /*...*/ };
  ~C() { delete n; }
  void Foo( /*... a param list ...*/ )  { n->Foo( /*...*/ ); }
  void Bar( /*... a param list ...*/ )  { n->Bar( /*...*/ ); }
  // etc.
};
```

50 of 67

Future: Unified Type System, Object Model

Arbitrary combinations of members and bases:

- Any type can contain members and/or base classes of any other type. Virtual dispatch etc. work as expected.
  - At most one base class may be of ref/value/mixed type.
- Overhead (regardless of mixing complexity, including deep inheritance with mixing and virtual overriding at each level):
  - For each object: At most one additional object.
  - For each virtual function call: At most one additional virtual function call.

Pure type:

- The declared type category, members, and bases are either all CLI, or all native.

Mixed type:

- Everything else. Examples:

```
ref class Ref : R, public N1, N2 { string s; };
class Native : I1, I2 { MessageQueue m; };
```

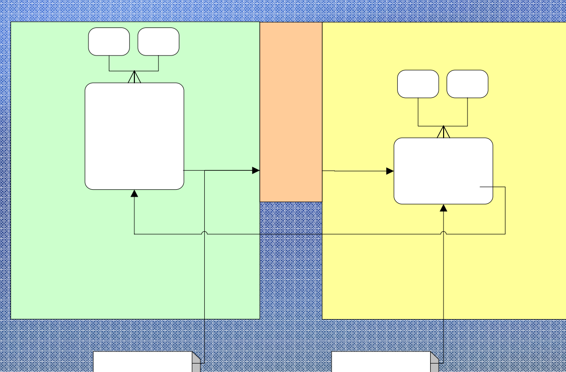
51 of 67

Future: Implementing Mixed Types

1 mixed = 1 pure + 1 pure.

```
ref class M : I1, I2, N1, N2 {
  System::String ^S1, ^S2;
  std::string s1, s2;
};
```

```
M* pm = new M;
M^ hm = gcnew M;
```



52 of 67

Future: Result for User Code

V1 Syntax:

```
private __gc class RadarForm : public Form {
  std::vector<RadarItem>* items;
  Native* n;

public:
  RadarForm() :
    : n( new Native )
    , items( new std::vector<RadarItem> )
    { /*...*/ };
  ~RadarForm() { delete items; delete n; }
  void Foo( /*... params ...*/ )
    { n->Foo( /*...*/ ); }
  void Bar( /*... params ...*/ )
    { n->Bar( /*...*/ ); }
  // etc.
};
```

V2 Syntax:

```
ref class RadarForm : Form, public Native {
  std::vector<RadarItem> items;
};
```

- One safe automated allocation, vs. *N* fragile handwritten allocations.
- This class is also better because it also has a destructor (implements IDisposable). That makes it work well by default with C++ automatic stack semantics (and C# using blocks, and VB/J# dispose patterns).

53 of 67

Other Features

Param arrays:

- Created when needed, preferred over varargs

```
void f( String^ str, ... array<Object^>^ arr );
f( "hello", 42, 3.14, "world" );
```

Unified CLI and C++ operators:

- Operators can now be static. Most work on handles.

```
ref class R { public: // ...
  static R^ operator+( R^ lhs, R^ rhs );
};
```

- Equality tests reference identity. Can be overridden by user.

Delegating constructors.

XML doc comments.

54 of 67

Pure Extensions to ISO C++

Only three reserved words:

gcnew generic nullptr

The rest are contextual keywords:

abstract delegate each event finally in initonly
interface literal override property ref sealed
value where

55 of 67

Implementation Details

Strategies for specifying contextual keywords:

- Spaced keywords: Courtesy Max Munch, Lex Hack & Assoc.
  for each enum class/struct interface class/struct
  ref class/struct value class/struct
- Contextual keywords that are never ambiguous: They appear in a grammar position where nothing may now appear.
  abstract finally in override sealed where
- Contextual keywords that can be ambiguous with identifiers: "If it can be an identifier, it is."
  delegate event initonly literal property
  Surgeon General's warning: Known to cause varying degrees of parser pain in compiler laboratory animals.

Not keywords, but in a namespace scope:

array interior_ptr pin_ptr safe_cast

56 of 67

Minimal Impact

Except for the three reserved words (and some macros), a well-formed program's meaning is unchanged.

Macro example #1:

```
// this has a different meaning in ISO C++ and C++/CLI
#define interface struct

// this has the same meaning in both
#define interface interface__
#define interface__ struct
```

Macro example #2:

```
// this has a different meaning in ISO C++ and C++/CLI
#define ref const
ref class C { } c;
```

57 of 67

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unified pointer and storage system (stack, native heap, gc heap).
   - Deterministic cleanup: Destruction/Dispose, finalization.
   - Generics × templates, STL on CLI.
   - Unified type system, mixing native/CLI, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

58 of 67

Why Standardize C++/CLI?

Primary motivators for C++/CLI standard:

- Stability of language.
- C++ community understands and demands standards.
- Openness promotes adoption.
- Independent implementations should interoperate.

Same TC39, new TG5: C++/CLI.

- C++/CLI is a binding between ISO C++ and ISO CLI only.
- Most of TG5's seven planned meetings are co-located with TG3 (CLI), and both standards are currently on the same schedule.

59 of 67

ISO and ECMA Structures

ISO SC22:

- WG3: APL
- WG4: Cobol
- WG5: Fortran
- WG9: Ada
- WG11: Binding techniques
- WG14: C
- WG15: POSIX
- WG16: Lisp
- WG17: Prolog
- WG19: Formal spec. langs.
- WG20: Internationalization
- WG21: C++

ECMA TC39:

- TG1: ECMAscript
- TG2: C#
- TG3: CLI
- TG4: Eiffel
- TG5: C++/CLI

60 of 67

The Importance of Bindings

Bindings for a language to other standards:

- Demonstrate that a language is important.
- Promote that language's use.

C has standardized bindings to important platforms:

- SQL (ISO SC32/WG3, ANSI/INCITS H2):
  - SQL/CLI (Client Level Interface) == ODBC. Antiquated. More safety and security issues than C++.
  - Around 1999, there was interest in both C++ and SQL to specify a C++ binding. Nothing happened.
- POSIX (ISO SC22/WG15):
  - A C API binding to an OS abstraction.
  - No longer under active development.

C++ doesn't, even though we've tried.

61 of 67

The Importance of Bindings (2)

Eiffel and C# have standardized bindings to CLI:

- Eiffel (ECMA TC39/TG4).
- C# (ECMA TC39/TG2).

C++ has to be a viable first-class language for CLI development:

- Key Q: "Why should a CLI developer use C++?"
- Key A: "Great leverage of C++ features and great CLI feature support" (not "imitate Eiffel or C#").
- Deliver promise of CLI.

OK, so it's good to make C++ support better. But why also standardize?

- To ensure independent implementations can interoperate.
- To ensure open participation.

62 of 67

C++/CLI Participants and Timeline

Participants:

- Convener: Tom Plum
- Project Editor: Rex Jaeschke
- Subject Matter Experts: Bjarne Stroustrup, Herb Sutter
- Participants: Dinkumware, EDG, Plum Hall, …
- Independent conformance test suite: Plum Hall

ECMA + ISO process, estimated timeline:

- Oct 1, 2003: ECMA TC39 plenary. Kick off TG5.
- Nov 15, 2003: Submit base document to ECMA.
  Microsoft will make this publicly available.
- Dec 2003 – Sep 2004: TG5 meetings (7).
- Dec 2004: Adopt ECMA standard.
- Dec 2004: Kick off ISO fast-track process.
- Dec 2005: Adopt ISO standard.

63 of 67

Draft TG5 Meeting Schedule

December 2003 – September 2004: TG5 meetings.

** = co-located with TC39/TG2/TG3*
*** = co-located with TC39/TG2/TG3 and adjacent to WG21*

	#	Dates	Location
	1	Dec 4-5, '03	College Station, TX
*	2	Jan 29-31, '04	Kona, HI
**	3	Mar 18-20, '04	Melbourne, Australia
	4	May 3-4, '04	Boston/NY/NJ area
*	5	Jun 14-15, '04	*tbd*
*	6	Aug 2-3, '04	*tbd: WA or OR*
*	7	Sep 21-22, '04	*tbd: Redmond, WA?*

64 of 67

Overview

1. Rationale and Goals
2. Language Tour
3. Design and Implementation Highlights
   - Unified pointer and storage system (stack, native heap, gc heap).
   - Deterministic cleanup: Destruction/Dispose, finalization.
   - Generics × templates, STL on CLI.
   - Unified type system, mixing native/CLI, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

65 of 67

Summary: C++ × CLI

C++ features:

- Deterministic cleanup, destructors.
- Templates.
- Native types.
- Multiple inheritance.
- STL, generic algorithms, lambda expressions.
- Pointer/pointee distinction.
- Copy construction, assignment.

CLI features:

- Garbage collection, finalizers.
- Generics.
- CLI types.
- Interfaces.
- Verifiability.
- Security.
- Properties, delegates, events.

66 of 67

Conclusion: The Two FAQs

Q: Is C++ relevant on modern VM / GC platforms?

- Heck, yeah.

Q: Why should a CLI programmer use C++?

- Preserves code base investment. Easiest migration for existing code base: "Just use /clr."
- Easiest and most efficient native interop, incl. mixed types.
- Deterministic (and automatic) cleanup as usual in C++, no coding patterns. Correctness by default.
- Leverage C++'s unique strengths (e.g., templates, generic programming, multiple inheritance, deterministic resource management and cleanup).
- Now not significantly harder or uglier than other languages.