The Future of C++ on .NET

A Tour of C++/CLI

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 .NET: 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 .NET platform.)
2. It's not hard to extend existing code with CLR 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 CLR.

2

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 CLR.
   - Mixing native/CLR, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

3

Microsoft's Bet on .NET

Windows XP SP1 (and onward): .NET ships in the OS

Windows Longhorn: .NET is the OS API (WinFX)

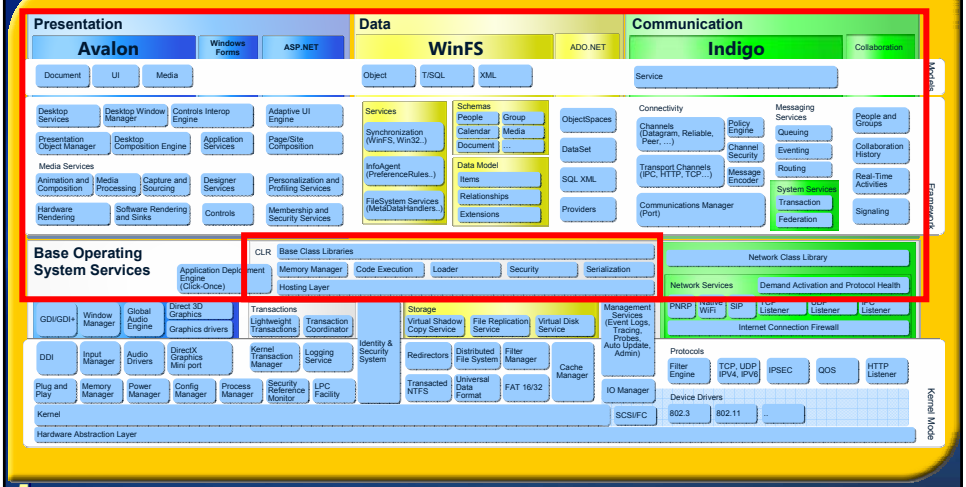
WinFX builds on the .NET Framework

Single cross-language framework for Windows

4

Microsoft's Bet on .NET

Windows Longhorn: .NET is the OS API (WinFX)



5

.NET (aka CLI) Cross-Platform

Microsoft .NET: WinFX, PocketPC, SPOT:

- Windows XP SP1 and onward: .NET ships in the OS.
- Windows Longhorn: .NET is the OS API (WinFX).
- PocketPC and SPOT devices (.NET Compact Framework).

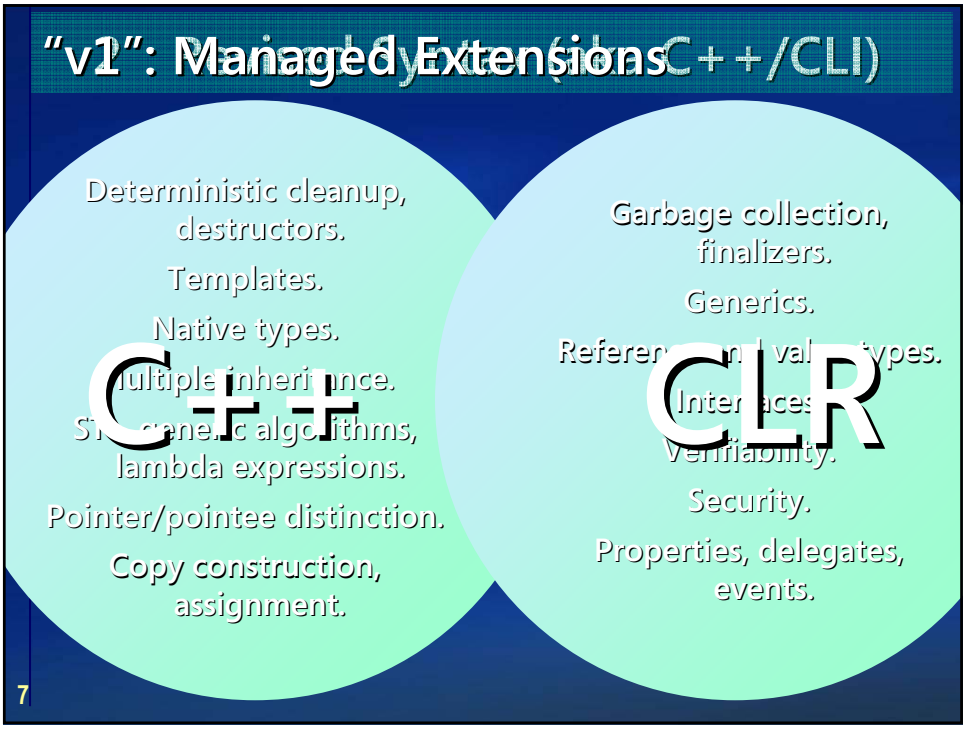
Microsoft Rotor:

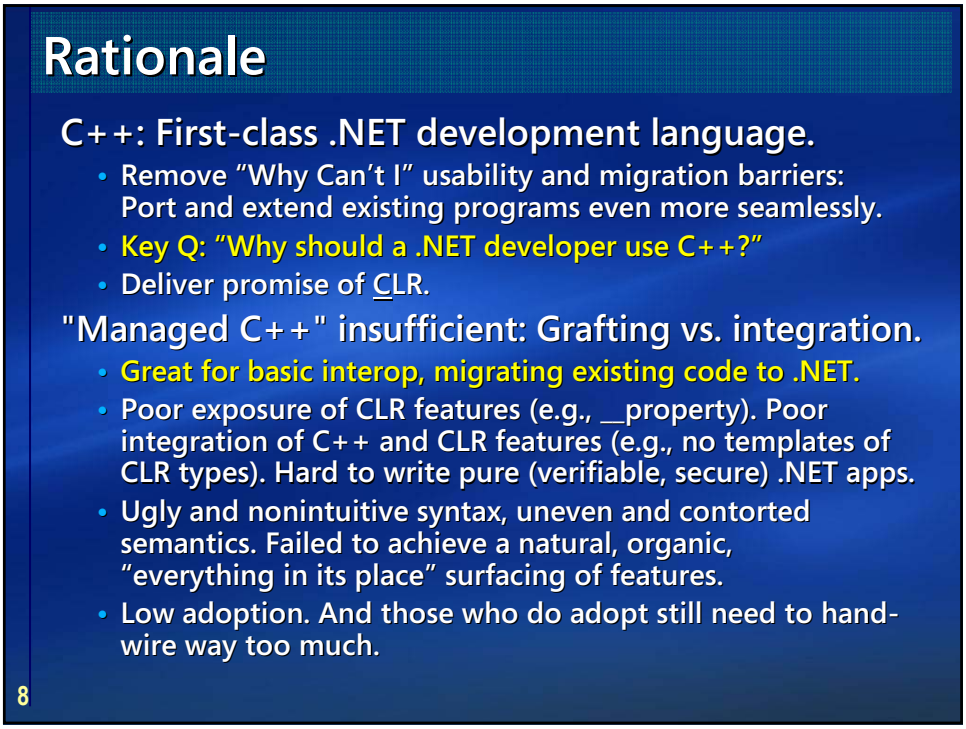
- SSCLI: Shared-source implementation of ISO CLI. For non-commercial use.
- Runs today on Windows XP, Mac OS X, and FreeBSD.

Novell (Ximian) Mono:

- First two commercial releases due Q2 2004 and Q4 2004.
- Runs on Unix and Linux. Targets x86 and PowerPC directly. Targets Arm, Sparc, HPPA, and s390 via interpreter.
- Includes ISO CLI, ASP.NET, ADO.NET, and GNOME- and Linux-specific libraries.

6


"v1": Managed Extensions (aka C++/CLI)
Deterministic cleanup, destructors.
Templates.
Native types.
Multiple inheritance.
lambda expressions.
Pointer/pointee distinction.
Copy construction, assignment.
C++
Garbage collection, finalizers.
Generics.
Verifiability.
Security.
Properties, delegates, events.
CLR
7


Rationale
C++: First-class .NET development language.
• Remove "Why Can't I" usability and migration barriers: Port and extend existing programs even more seamlessly.
• Key Q: "Why should a .NET developer use C++?"
• Deliver promise of CLR.
"Managed C++" insufficient: Grafting vs. integration.
• Great for basic interop, migrating existing code to .NET.
• Poor exposure of CLR features (e.g., __property). Poor integration of C++ and CLR features (e.g., no templates of CLR types). Hard to write pure (verifiable, secure) .NET 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.
8

Major Goals

Feature coverage:

- Provide organic support for CLR 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++.

C++ × CLR: Why a .NET programmer should use C++.

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

9

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 CLR.
   - Mixing native/CLR, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

10

adjective class C;

11

Basic Class Declaration Syntax

Type are declared "*adjective* class":

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

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

Examples:

```
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
```

12

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
```

13

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()
```

14

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 );
  }
};
```

15

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.

16

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 class R : I1, I2, I3 {
public:
  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
};
```

17

Delegating Constructors

Can delegate to one peer constructor. No cycle detection is required.

```
ref class R {
  S s;
  T t;
  R( int i, const U& u ) : s(i), t(u) { /* init */ }
public:
  R()          : R( 42,  3.14 ) { }
  R( int i )   : R(  i,  3.14 ) { }
  R( U& u ) : R( 53,     u ) { }
};
```

18

CLR 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
```

19

Other Features

Param arrays:

- Created when needed, preferred over varargs

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

Unified CLR 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.

XML doc comments.

20

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 CLR.
   - Mixing native/CLR, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

21

% is to ^
as
& is to *

22

Unified Storage/Pointer Model

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

- On the native heap: `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: `T^ t2 = gcnew T;`
  - Handles (^) are object references (to whole objects).
  - Calling delete is optional: "Destroy now, or finalize later."
- On the stack, 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

Pointers and Handles

Native pointers (*) and handles (^):

- ^ is like *. Differences: ^ points to a whole object on the gc heap, 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

References and Unary &/%

Native (&) and tracking (%) references:

- % is like &. Differences: % can refer into any memory area incl. the gc heap. 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>, when myobj is physically on the gc heap).
- %myobj → MyType^.

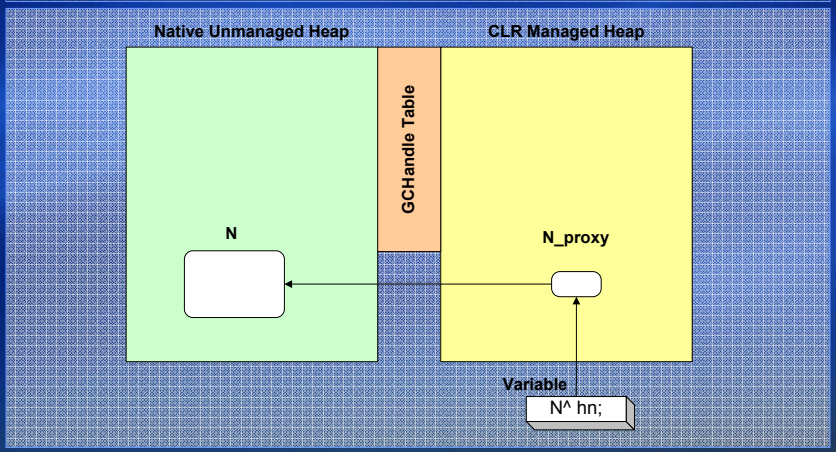
25

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
```



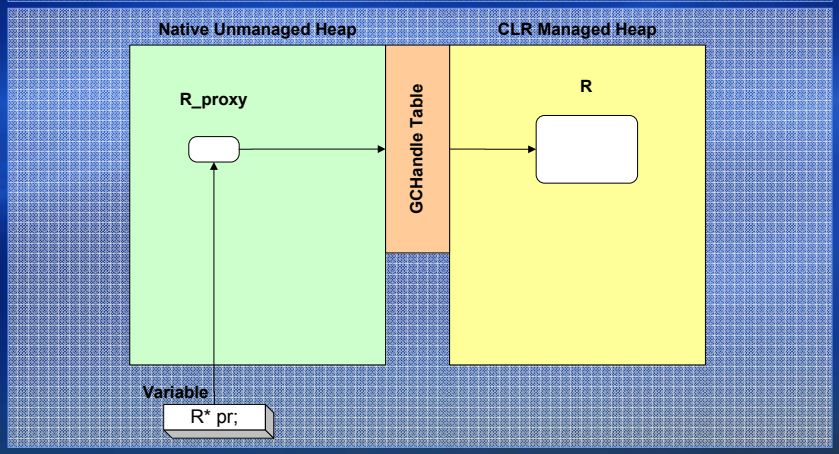
26

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
```

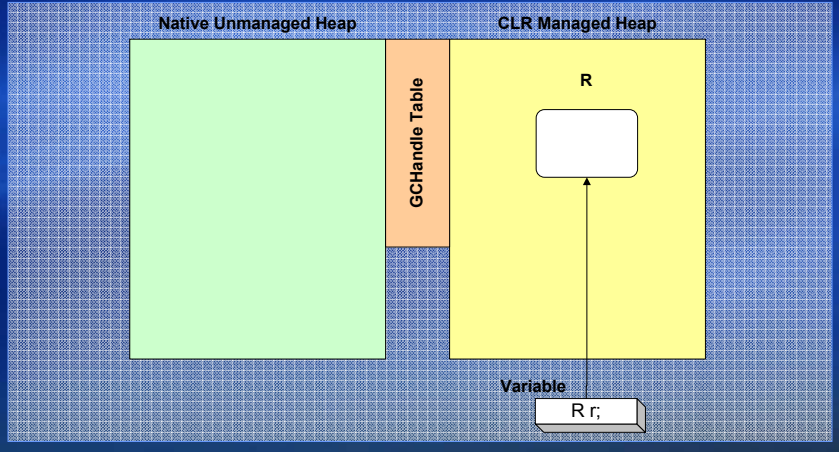


27

Ref Class on the Stack

The type of "%R" is R^.

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



28

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

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*
```

30

* assuming copy construction/assignment are defined

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 CLR.
   - Mixing native/CLR, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

31

T:: ~T()
and
T:: !T()

32

Cleanup in C++: Less Code, More Control

The CLR 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

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

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

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

Deterministic Cleanup in VB/Java

Alternative equivalent (in C# syntax):

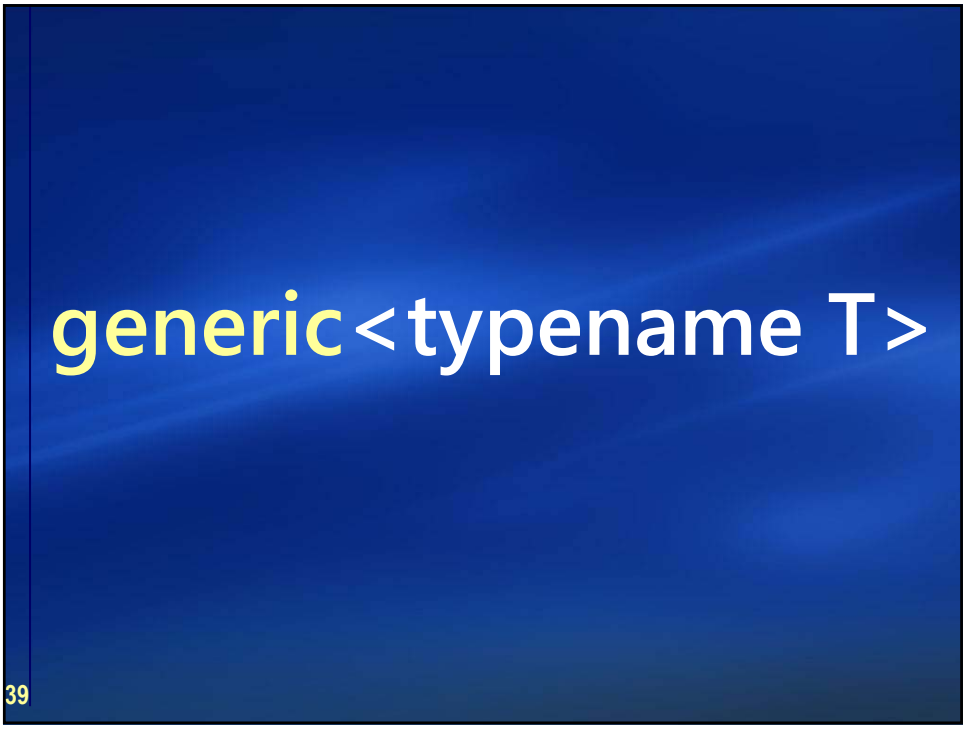
```
void Transfer() {
  MessageQueue source = null, dest1 = null, dest2 = null;
  try {
    source = new MessageQueue( "server\\sourceQueue" );
    String qname = (String)source.Receive().Body;
    dest1 = new MessageQueue( "server\\" + qname );
    dest2 = new MessageQueue( "backup\\" + qname );
    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

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 CLR.
   - Mixing native/CLR, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

38



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).

40

Generics

Generics are declared much like templates:

```
generic<typename T>
where T : IDisposable, IFoo
ref class GR { /* … */ };
```

- 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
```

41

STL on the CLR

C++ enables STL on CLR:

- 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( String^ s in v ) Console::WriteLine( s );
```

42

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 CLR.
   - Mixing native/CLR, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

43

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

44

CLR 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 );
  }
};
```

45

CLR 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
```

46

Native Types in the CLR 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
```

47

What Customers 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.

48

What Customers Are Doing (2)

Example 2: Faking up base classes
(e.g., expose native types to a CLR 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.
};
```

49

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 CLR, or all native.

Mixed type:

- Everything else. Examples:

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

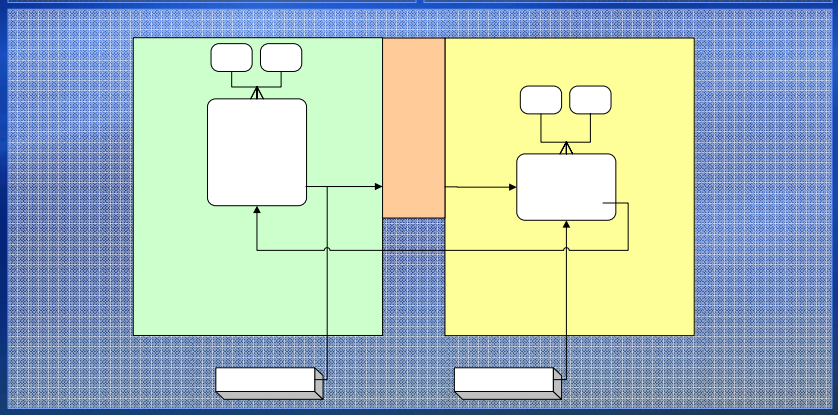
50

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;
```



51

Future: Result for Customer 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).

52

Native Unmanaged He

N1 N2

N1 base1
N2 base2
string s1
string s2
nt32
part

Variable
M* pm

Pure Extensions to ISO C++

Only three reserved words:

gcnew generic nullptr

The rest are contextual keywords:

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

53

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 CLR.
   - Mixing native/CLR, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

54

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.

55

C++/CLI Participants and Timeline

Participants:

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

Ecma + ISO process, estimated timeline:

- Oct 1, 2003: Ecma TC39 plenary. Kicked off TG5.
- Nov 21, 2003: Submitted base document to Ecma.
- 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.

56

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 CLR.
   - Mixing native/CLR, other features.
4. C++/CLI Standardization
   - Venue, players, timelines, how to participate.

57

Summary: C++ × CLR

C++ features:

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

CLR features:

- Garbage collection, finalizers.
- Generics.
- Reference and value types.
- Interfaces.
- Verifiability.
- Security.
- Properties, delegates, events.

58

Conclusion: The Two FAQs

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

- Heck, yeah.

Q: Why should a .NET 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.

59