Is C++ Relevant on Modern Environments?

A Tour of C++/CLI

Herb Sutter

Architect
Microsoft Visual C++

Overview

Rationale and Goals

Language Tour

Design and Implementation Highlights

- **Unified pointer and storage system (stack, native heap, gc heap).**
- **Deterministic cleanup: Destruction/Dispose, finalization.**
- **Generics × templates, STL on CLR.**

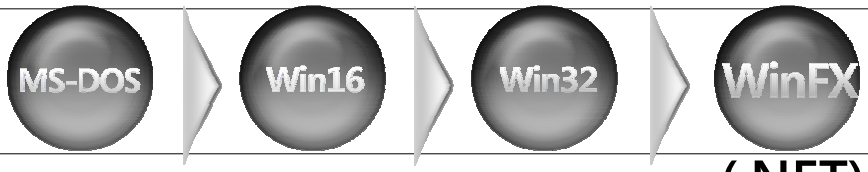
Standardization and Futures

2

Microsoft's Bet on .NET

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

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



(.NET)

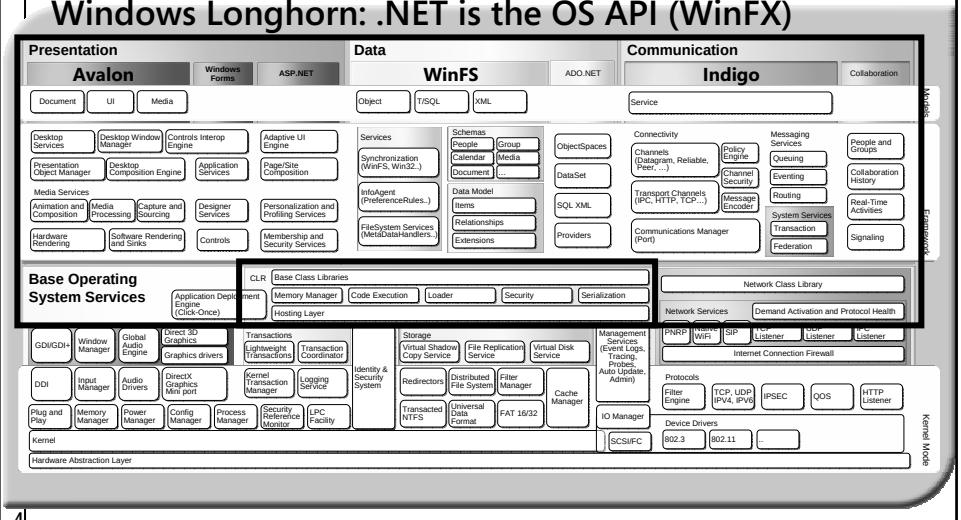
WinFX builds on the .NET Framework

Single cross-language framework for Windows

3

Microsoft's Bet on .NET

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



4

.NET (aka ISO CLI) Cross-Platform

Microsoft .NET: WinFX, PocketPC, SPOT:

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

Microsoft Rotor, shared source:

- Shared-source implementation for non-commercial use. Runs today on Windows XP, Mac OS X, and FreeBSD.

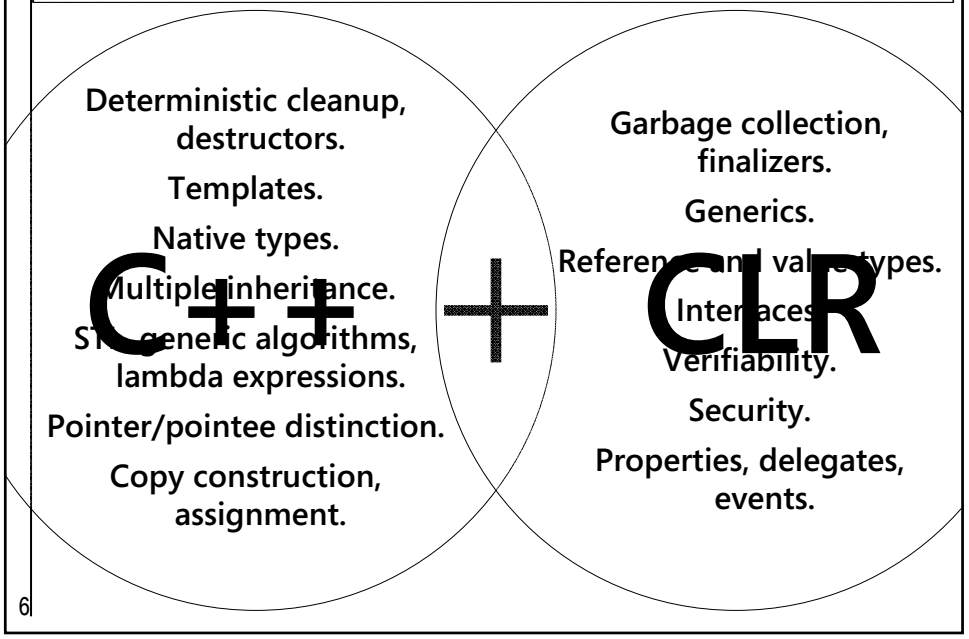
> Quote of the day: "You may modify this Software and distribute the modified Software for non-commercial purposes"

Novell Mono, GPL open source (*go-mono.org*):

- First two commercial releases due Q2 2004 and Q4 2004.
- Runs on Unix and Linux. Includes ISO CLI, ASP.NET, ADO.NET, and GNOME- and Linux-specific libraries.

5

2004/5: C++ in Whidbey



6

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

Why a Language-Level Binding

Garbage collection requires new pointer abstractions:

- Plain ISO C++ pointers/references can support some GC, imperfectly but acceptably for a wide class of applications (e.g., Boehm, a conservative collector).
- ISO C++ pointers/references cannot support general GC. Compacting GCs move objects, which means pointer values can change, pointers can't be ordered, pointers can't be hidden by casting to int and back, etc.

Type system differences are small, but need to be surfaced in type declarations:

- All CLI types: Deep virtual calls in constructors.
- CLI value types: Hybrid boxed/unboxed form.
- CLI interfaces: A lot like normal C++ abstract virtual base classes, but subtle differences.

8

Overview

Rationale and Goals

Language Tour

Design and Implementation Highlights

- Unified pointer and storage system (stack, native heap, gc heap).
- Deterministic cleanup: Destruction/Dispose, finalization.
- Generics × templates, STL on CLR.

Standardization and Futures

9

adjective class C;

10

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++ & CLR 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
```

11

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
```

12

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::set( String^s, 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()
```

13

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.

14

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
};
```

15

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 ) { }
};
```

16

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
```

17

Overview

Rationale and Goals

Language Tour

Design and Implementation Highlights

- Unified pointer and storage system (stack, native heap, gc heap).
- Deterministic cleanup: Destruction/Dispose, finalization.
- Generics × templates, STL on CLR.

Standardization and Futures

18

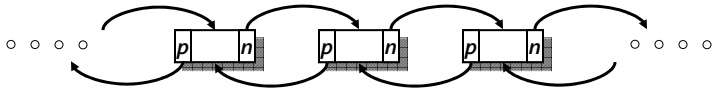
Stupid Pointer Tricks (or, Why a new abstraction?)

C/C++ lets you do (way too?) much with pointers:

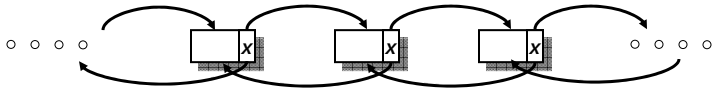
- Hide them: Cast to int/void* and back. Write them to disk.
- Order them. Example: *set*<*int**>.
- XOR them.

"XOR them?!" No, really, this is an actual technique.

- Consider a traditional bidirectional list (overhead = two pointers per node):



- Instead of storing two pointers, store one: *x* = prev xor next.



- When traversing, remember the node you came from, *n*. Regardless of direction, the next node's address is *x xor &n*.

19

% is to ^

as

& is to *

20

Storage and Pointer Model

Create objects on the native heap, CLR (gc) heap, or on the stack:

- On the native heap (native types): `T* t1 = new T;`
  - As usual, pointers (*) are stable, even during GC. 100% backward compatible with all standard and nonstandard stupid pointer tricks.
  - As usual, failure to explicitly call *delete* will leak.
- On the gc heap (CLR types): `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: Deterministic destruction/dispose is automatic and implicit, hooked to stack unwinding or to the enclosing object's lifetime.

21

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* )
```

22

Overview

Rationale and Goals

Language Tour

Design and Implementation Highlights

- Unified pointer and storage system (stack, native heap, gc heap).
- Deterministic cleanup: Destruction/Dispose, finalization.
- Generics × templates, STL on CLR.

Standardization and Futures

24

Why it's important to know "why": A parable.

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.
  Other languages: Correctness by explicit coding.

26

T::~T()
destroy now

or

T:: !T()
finalize later

27

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.

28

Determinism Matters: Performance

Microsoft.com live experience (~7/15/2003):

- Too many mid-life objects leaking into Gen2 caused frequent full collections. Result: 70% time in GC.
- CLR performance team suggested changes to release as many objects as possible before making server to server calls. Result: ~1% time in GC after the fix.

29

Side By Side: Using a StreamReader

C++

```
String^ ReadFirstLineFromFile( String^ path ) {
  StreamReader r(path);
  return r.ReadLine();
}
```

C#

```
String ReadFirstLineFromFile( String path ) {
  using ( StreamReader r = new StreamReader(path) ) {
    return r.ReadLine();
  }
}
```

Java

```
String ReadFirstLineFromFile( String path ) {
  StreamReader r = null;
  String s = null;
  try {
    r = new StreamReader(path);
    s = r.ReadLine();
  } finally {
    if ( r != null ) r.Dispose();
  }
  return s;
}
```

30

Side By Side: Using "lock"

C++

```
{
  lock l( obj );
  ... do something with shared state ...
}
```

C#

```
lock( obj ) {
  ... do something with shared state ...
}
```

Java

```
Monitor.Enter( obj );
try {
  ... do something with shared state ...
} finally {
  Monitor.Exit( obj );
}
```

31

Side By Side: Nontrivial "lock"

C++

```
{
  lock l( obj, 10 );
  ... do something with shared state ...
}
```

C#

```
if( !Monitor.TryEnter( obj, TimeSpan.FromSeconds( 10 ) ) ) {
  throw new Something();
}
try {
  ... do something with shared state ...
} finally {
  Monitor.Exit( obj );
}
```

Java

```
if( !Monitor.TryEnter( obj, TimeSpan.FromSeconds( 10 ) ) ) {
  throw new Something();
}
try {
  ... do something with shared state ...
} finally {
  Monitor.Exit( obj );
}
```

32

Side By Side: Nontrivial "lock"

C++

```
{
  lock l( obj, 10 );
  ... do something with shared state ...
}
```

C#

```
using( new Lock( obj, 10 ) ) {
  ... do something with shared state ...
}
```

```
public class Lock
  : IDisposable {
 private object target;
 public Lock(
    object o, double tm
    ) {
    target = o;
    if( !Monitor.TryEnter( o, tm ) )
      throw new Something();
  }
```

```
void IDisposable.Dispose() {
    Monitor.Exit( target );
#if DEBUG
    GC.SuppressFinalize( this );
#endif
  }
#if DEBUG
  ~Lock() {
    Diagnostics.Debug.Fail(
      "Undisposed lock"
    );
  }
#endif
}
```

33

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.

34

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 );
    }
  }
}
```

35

Deterministic Cleanup in VB/Java

Minimal Java equivalent:

```
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(); }
  }
}
```

36

Overview

Rationale and Goals

Language Tour

Design and Implementation Highlights

- Unified pointer and storage system (stack, native heap, gc heap).
- Deterministic cleanup: Destruction/Dispose, finalization.
- Generics × templates, STL on CLR.

Standardization and Futures

37

generic<typename T>

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.

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

- Constraints are inheritance-based.

Templates:

- Compile-time, C++, and generally intra-assembly.
- Not a high burden: 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).

39

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 );
```

40

Overview

Rationale and Goals

Language Tour

Design and Implementation Highlights

- Unified pointer and storage system (stack, native heap, gc heap).
- Deterministic cleanup: Destruction/Dispose, finalization.
- Generics × templates, STL on CLR.

Standardization and Futures

41

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.

42

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: Vote on whether to adopt as Ecma standard.
- Q1 2005: If successful, submit for ISO fast-track process.
- Q1 2006: If ready, vote on whether to adopt ISO standard.

43

Future: Unify Memory and Object Models

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

- On the native heap (any type): `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 (any type): `T^ t2 = gcnew T;`
  - Handles (^) are object references (to whole objects).
  - Calling delete is optional: "Destroy now, or finalize later."
- On the stack: `T t3;`
  - Q: Why would you? A: Deterministic destruction/dispose is automatic and implicit, hooked to stack unwinding or to the enclosing object's lifetime.

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.

44

Overview

Rationale and Goals

Language Tour

Design and Implementation Highlights

- Unified pointer and storage system (stack, native heap, gc heap).
- Deterministic cleanup: Destruction/Dispose, finalization.
- Generics × templates, STL on CLR.

Standardization and Futures

45

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.

46

Choosing a Language

Q: Which .NET language should you use?

Microsoft's answer, in our Whidbey release:

- C++ is the recommended path to .NET and Longhorn.
- *If you have an existing C++ code base:* Keep using C++.
- *If you want to make frequent use of native code/libs:* Use C++. It's far simpler (seamless) and faster.
- *If you want to write brand-new pure-.NET apps that rarely or never interop with native code:* Use whatever language you're already comfortable with, and pick based on language features. The .NET features and Frameworks library are available equally through all languages.

47

A Word About Free Software

Announcing Visual C++ Toolkit 2003:

- When: In four days (Monday, 19 April 2004).
- Where: *http://msdn.microsoft.com/visualc* .
- What: The VC++ command-line compiler with full optimizing compiler and C and C++ standard libraries. (Doesn't include the IDE or MFC, for example.)
  - FAQ #1: Yes, it's the most-current and -conforming version of the compiler (VC++ 2003, a.k.a. VC++ 7.1).
  - FAQ #2: Yes, it's free of restrictions: No time limits, no limits on what you can build with it (both commercial and noncommercial use are fine).
- Why: Standards are a good thing. So having more conforming compilers freely available for people to try and use is a good thing too.

48

Is C++ Relevant on Modern Environments?

A Tour of C++/CLI

Questions?