



Remedy IT

Your challenge - our solution

Comparing IDL to C++ with IDL to C++11

Simplify development of CORBA, DDS, and CCM based applications



Remedy IT

Your challenge - our solution

Overview

- This presentations gives a comparison between the IDL to C++ and IDL to C++11 language mappings
- It assumes basic understanding of IDL and CORBA
- For more information take a look at our TAOX11 website at <https://taox11.remedy.nl>



Remedy IT

Your challenge - our solution

Introduction



Problems with IDL to C++

- The IDL to C++ language mapping is from the 90's
- IDL to C++ could not depend on various C++ features as
 - C++ namespace
 - C++ exceptions
 - Standard Template Library
- As a result
 - Mapping is hard to use correctly
 - Uses its own constructs for everything



Why a new language mapping?

- IDL to C++ language mapping is impossible to change because
 - Multiple implementations are on the market (open source and commercial)
 - A huge amount of applications have been developed
- An updated IDL to C++ language mapping would force all vendors and users to update their products
- The standardization of a new C++ revision in 2011 (ISO/IEC 14882:2011, called C++11) gives the opportunity to define a new language mapping
 - C++11 features are not backward compatible with C++03 or C++99
 - A new C++11 mapping leaves the existing mapping intact



Goals

Remedy IT

Your challenge - our solution

- Simplify mapping for C++
- Make use of the new C++11 features to
 - Reduce amount of application code
 - Reduce amount of possible errors made
 - Gain runtime performance
 - Speedup development and testing
 - Faster time to market
 - Reduced costs
 - Reduced training time



Remedy IT

Your challenge - our solution

OMG Specification

- IDL to C++11 v1.3 available from the OMG website at <http://www.omg.org/spec/CPP11/>
- [Revision Task Force](#) (RTF) is active to work on issues reported



Remedy IT

Your challenge - our solution

IDL Constructs



Modules

Remedy IT

Your challenge - our solution

An IDL module maps to a C++ namespace with the same name, same for both mappings

IDL

```
module M
{
  // definitions
};

module A
{
  module B
  {
    // definitions
  };
};
```

C++/C++11

```
namespace M
{
  // definitions
};

namespace A
{
  namespace B
  {
    // definitions
  };
};
```



Basic types

Remedy IT

Your challenge - our solution

| IDL | C++ | C++11 | C++11 Default value |
|--------------------|-------------------|-------------|---------------------|
| short | CORBA::Short | int16_t | 0 |
| long | CORBA::Long | int32_t | 0 |
| long long | CORBA::LongLong | int64_t | 0 |
| unsigned short | CORBA::UShort | uint16_t | 0 |
| unsigned long | CORBA::ULong | uint32_t | 0 |
| unsigned long long | CORBA::ULongLong | uint64_t | 0 |
| float | CORBA::Float | float | 0.0 |
| double | CORBA::Double | double | 0.0 |
| long double | CORBA::LongDouble | long double | 0.0 |
| char | CORBA::Char | char | 0 |
| wchar | CORBA::WChar | wchar_t | 0 |
| boolean | CORBA::Boolean | bool | false |
| octet | CORBA::Octet | uint8_t | 0 |



Constants

Remedy IT

Your challenge - our solution

```
const string name = "testing";  
interface A  
{  
    const float pi = 3.14159;  
};
```

C++

```
const char *const name = "testing";  
  
class A  
{  
public:  
    static const CORBA::Float pi;  
};
```

C++11

```
const std::string name {"testing"};  
  
class A  
{  
public:  
    static constexpr float pi {3.14159F};  
};
```



String types

Remedy IT

Your challenge - our solution

```
string name;  
wstring w_name;
```

C++

```
CORBA::String_var name;  
CORBA::WString_var w_name;  
name = CORBA::string_dup ("Hello");  
std::cout << name.in () << std::endl;
```

C++11

```
std::string name {"Hello"};  
std::wstring w_name;  
std::cout << name << std::endl;
```



Enum

Remedy IT

Your challenge - our solution

```
enum Color {  
    red,  
    green,  
    blue  
};
```

C++

```
enum Color  
{  
    red,  
    green,  
    blue  
};  
  
Color mycolor  
mycolor = red;  
if (mycolor == red)  
{  
    std::cout << "Correct color";  
}
```

C++11

```
enum class Color : uint32_t  
{  
    red,  
    green,  
    blue  
};  
  
Color mycolor {Color::red};  
if (mycolor == Color::red)  
{  
    std::cout << "Correct color";  
}
```



Sequence

Remedy IT

Your challenge - our solution

```
typedef sequence<long> LongSeq;
```

C++

```
LongSeq mysequence;  
  
// Add an element to the vector  
Mysequence.length (1)  
Mysequence[1] = 5;  
  
for (CORBA::ULong i = 0; I <  
     mysequence.length(); i++)  
{  
    std::cout << mysequence[i] << ";" <<  
    std::endl;  
}
```

C++11

```
LongSeq mysequence;  
  
// Add an element to the vector  
mysequence.push_back (5);  
  
// Dump using C++11 range based for loop  
for (const int32_t& e : mysequence)  
{  
    std::cout << e << ";" << std::endl;  
}
```



Struct (1)

Remedy IT

Your challenge - our solution

```
struct Variable {  
    string name;  
};
```

C++

```
struct Variable {  
    string name;  
};
```

C++11

```
class Variable final  
{  
public:  
    Variable ();  
    ~Variable ();  
    Variable (const Variable&);  
    Variable (Variable&&);  
    Variable& operator= (const Variable& x);  
    Variable& operator= (Variable&& x);  
    explicit Variable (std::string name);  
    void name (const std::string& _name);  
    void name (std::string&& _name);  
    const std::string& name () const;  
    std::string& name ();  
};  
  
namespace std  
{  
    template <>  
    void swap (Variable& m1, Variable& m2);  
};
```



Struct (2)

Remedy IT

Your challenge - our solution

C++

```
Variable v;  
Variable v2 ("Hello");  
CORBA::String_var myname =  
    CORBA::String_dup ("Hello");  
  
// Set a struct member  
v.name = CORBA::String_dup (myname.in ());  
  
// Get a struct member  
CORBA::String_var l_name =  
    CORBA::String_dup (v.name.in ());  
std::cout << "name" << l_name.in () <<  
    std::endl;  
  
if (strcmp (v.in (), v2.in () != 0)  
{  
    std::cerr << "names are different"  
    <<std::endl;  
}
```

C++11

```
Variable v;  
Variable v2 ("Hello");  
std::string myname {"Hello"};  
  
// Set a struct member  
v.name (myname);  
  
// Get a struct member  
std::cout << "name" << v.name () <<  
    std::endl;  
  
if (v != v2)  
{  
    std::cerr << "names are different"  
    <<std::endl;  
}
```




C++11 Reference types (1)

- An IDL interface maps to so called reference types
- Reference types are reference counted, for example given type A
 - Strong reference type behaves like `std::shared_ptr` and is available as `IDL::traits<A>::ref_type`
 - Weak reference type behaves like `std::weak_ptr` and is available as `IDL::traits<A>::weak_ref_type`
- A nil reference type is represented as `nullptr`
- Invoking an operation on a nil reference results in a `INV_OBJREF` exception



C++11 Reference types (2)

Remedy IT

Your challenge - our solution

**Given IDL type A the mapping delivers
IDL::traits<A> with type traits**

IDL

```
interface A
{
    // definitions
};
```

C++11

```
// Obtain a reference
IDL::traits<A>::ref_type a = // .. obtain a
    // reference

// Obtain a weak reference
IDL::traits<A>::weak_ref_type w =
    a.weak_reference();

// Obtain a strong reference from a weak one
IDL::traits<A>::ref_type p = w.lock ();

if (a == nullptr) // Legal comparisons
if (a != nullptr) // legal comparison
if (a) // legal usage, true if a != nullptr
if (!a) // legal usage, true if a == nullptr
if (a == 0) // illegal, results in a compile
    // error
delete a; // illegal, results in a compile error
```



C++11 Reference types (2)

Remedy IT

Your challenge - our solution

Reference types can only be constructed using CORBA::make_reference

IDL

```
interface A
{
    // definitions
};
```

C++11

```
// Servant implementation class
class A_impl final :
    CORBA::servant_traits<A>::base_type
{
}

// Create a servant reference using
// make_reference
CORBA::servant_traits<A>::ref_type a_ref =
    CORBA::make_reference<A_impl> ();

// We could use new, but the resulting
// pointer can't be used for making any
// CORBA call because the pointer can't be
// used to construct a reference type which
// is the only thing the API accepts
A_impl* p = new ACE_impl ();

// Or we can obtain a reference from another
// method
IDL::traits<A>::ref_type = foo->get_a ();
```



C++11 Reference types (3)

Remedy IT

Your challenge - our solution

Widening and narrowing references

IDL

```
interface A
{
    // definitions
};

interface B : A
{
    // definitions
};
```

C++11

```
IDL::traits<B>::ref_type bp = ...

// Implicit widening
IDL::traits<A>::ref_type ap = bp;

// Implicit widening
IDL::traits<Object>::ref_type objp = bp;

// Implicit widening
objp = ap;

// Explicit narrowing
bp = IDL::traits<B>::narrow (ap)
```



Reference types

Remedy IT

Your challenge - our solution

Using a reference type

C++

```
class B {
public:
    // Store the reference in a member
    B (A_ptr a) : a_(A::_duplicate (a)) {}
    // Return a reference
    A_ptr get_A () { return A::_duplicate
        (a_.in ()); }
private:
    A_var a_;
};
```

C++11

```
class B {
public:
    // Store the reference in a member
    B (IDL::ref_type<A> a) : a_ (a)
    // Return a reference
    IDL::ref_type<A> get_A() { return a_;}
private:
    IDL::ref_type<A> a_;
};
```



Argument passing

- Simplified rules for argument passing compared to IDL to C++
- No need for new/delete when passing arguments
- The C++11 move semantics can be used to prevent copying of data
- Given an argument of A of type P:
 - In: for all primitive types, enums, and reference types, the argument is passed as P. For all other types, the argument is passed as const P&
 - Inout: passed as P&
 - Out: passed as P&
 - Return type: returned as P



Interfaces implementation

- Given a local interface A the implementation has to be derived from `IDL::traits<A>::base_type`
- Given a regular interface A the CORBA servant implementation has to be derived from `CORBA::servant_traits<A>::base_type`
- In both cases a client reference is available as `IDL::traits<A>::ref_type`



Implementing a servant

Remedy IT

Your challenge - our solution

Implement a CORBA servant for interface A

C++

```
class A_impl : public virtual  
    POA::A  
{  
};
```

C++11

```
class A_impl : public virtual  
    CORBA::servant_traits<A>::ref_type  
{  
};
```




Remedy IT

Your challenge - our solution

C++11 Example application



CORBA Hello world

Remedy IT

Your challenge - our solution

IDL

```
interface Hello
{
    /// Return a simple string
    string get_string ();

    /// A method to shutdown the server
    oneway void shutdown ();
};
```

Copyright © Remedy IT



CORBA client

Remedy IT

Your challenge - our solution

```
int main(int argc, char* argv[])
{
    try
    {
        // Obtain the ORB
        IDL::traits<CORBA::ORB>::ref_type orb = CORBA::ORB_init (argc, argv);

        // Create the object reference
        IDL::traits<CORBA::Object>::ref_type obj = orb->string_to_object ("file://test.ior");

        // Narrow it to the needed type
        IDL::traits<Hello>::ref_type hello = IDL::traits<Hello>::narrow (obj);

        // Invoke a method, invoking on a nil reference will result in an exception
        std::cout << "hello->get_string () returned " << hello->get_string () << std::endl;

        // Shutdown the server
        hello->shutdown ();

        // Cleanup our ORB
        orb->destroy ();
    }
    catch (const std::exception& e)
    {
        // All exceptions are derived from std::exception
        std::cerr << "exception caught: " << e.what () << std::endl;
    }
    return 0;
}
```



CORBA servant

Remedy IT

Your challenge - our solution

- C++11 CORBA servant for type T must be derived from `CORBA::servant_traits<T>::base_type`

```
class Hello final : public virtual CORBA::servant_traits<Hello>::base_type
{
public:
    // Constructor
    Hello (IDL::traits<CORBA::ORB>::ref_type orb) : orb_ (orb) {}
    // Destructor
    virtual ~Hello () {}
    // Implement pure virtual methods from the base_type
    virtual std::string get_string () override
    {
        return "Hello!";
    }
    virtual void shutdown () override
    {
        this->orb_->shutdown (false);
    }
private:
    // Use an ORB reference to shutdown the application.
    IDL::traits<CORBA::ORB>::ref_type orb_;
};
```



CORBA server (1)

Remedy IT

Your challenge - our solution

```
int main(int argc, char* argv[])
{
    try
    {
        // Obtain our ORB
        IDL::traits<CORBA::ORB>::ref_type orb = CORBA::ORB_init (argc, argv);

        // Obtain our POA and POAManager
        IDL::traits<CORBA::Object>::ref_type obj = orb->resolve_initial_references ("RootPOA");
        IDL::traits<PortableServer::POA>::ref_type root_poa =
            IDL::traits<PortableServer::POA>::narrow (obj);
        IDL::traits<PortableServer::POAManager>::ref_type poaman = root_poa->the_POAManager ();

        // Create the servant
        CORBA::servant_traits<Hello>::ref_type hello_impl =
            CORBA::make_reference<Hello> (orb);

        // Activate the servant as CORBA object
        PortableServer::ObjectId id = root_poa->activate_object (hello_impl);
        IDL::traits<CORBA::Object>::ref_type hello_obj = root_poa->id_to_reference (id);
        IDL::traits<Hello>::ref_type hello =
            IDL::traits<Hello>::narrow (hello_obj);

        // Put the IOR on disk
        std::string ior = orb->object_to_string (hello);
        std::ofstream fos("test.ior");
        fos << ior;
        fos.close ();
    }
}
```



CORBA server (2)

Remedy IT

Your challenge - our solution

```
// Activate our POA
poaman->activate ();

// And run the ORB, this method will return at the moment the ORB has been shutdown
orb->run ();

// Cleanup our resources
root_poa->destroy (true, true);
orb->destroy ();
}
catch (const std::exception& e)
{
    // Any exception will be caught here
    std::cerr << "exception caught: " << e.what () << std::endl;
}

return 0;
}
```



Tips & Tricks

Remedy IT

Your challenge - our solution

- Don't use new/delete
- Use pass by value together with C++11 move semantics



Conclusion

- C++11 simplifies CORBA programming
- The combination of reference counting and C++11 move semantics make the code much safer and secure
- Application code is much smaller and easier to read



TAOX11

Remedy IT

Your challenge - our solution

- Commercial CORBA implementation from Remedy IT
- Compliant with IDL to C++11 v1.3
- IDL compiler with front end supporting IDL2, IDL3, and IDL3+
- Free evaluation versions available from <https://swsupport.remedy.nl/>!
- More details at <https://taox11.remedy.nl/>



Remedy IT

Your challenge - our solution

Want to know more?

- Look at TAOX11 at <https://taox11.remedy.nl/>
- Check the Remedy IT provided examples at <https://github.com/RemedyIT/idl2cpp11>
- Contact us, see <https://www.remedy.nl/>



Contact

Remedy IT

Your challenge - our solution

Remedy IT
The Netherlands

tel.: +31(0)88 – 053 0000

e-mail: sales@remedy.nl

website: <https://www.remedy.nl/>

Twitter: [@RemedyIT](https://twitter.com/RemedyIT)

Slideshare: [RemedyIT](#)

Subscribe to our [mailing list](#)