



Remedy IT

Your challenge - our solution

CORBA Programming with TAOX11

The C++11 CORBA Implementation





TAOX11: the CORBA Implementation by Remedy IT

- [TAOX11](#) simplifies development of CORBA based applications
- IDL to C++11 language mapping is easy to use
- Greatly reduces the CORBA learning curve
- Reduces common user mistakes
- Improves application stability and reliability
- Significant improvement of run-time performance
- CORBA AMI support
- Extended suite of unit tests and examples



TAOX11

Remedy IT

Your challenge - our solution

- Commercial CORBA implementation by Remedy IT
- Compliant with the OMG IDL to C++11 language mapping
- IDL compiler with front end supporting IDL2, IDL3, and IDL3+
- Free evaluation versions available from our [software support portal](#) !
- More details on <http://taox11.remedy.nl>



Remedy IT

Your challenge - our solution

Tutorial overview

- This tutorial gives an overview of the IDL to C++11 language mapping
- Introduces TAOX11, the C++11 CORBA implementation by Remedy IT
- It assumes basic understanding of IDL and CORBA



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 the IDL to C++ language 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 of IDL to C++11

- 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



OMG Specification

- IDL to C++11 v1.2 available from the OMG website as formal/2015-08-01
- [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

IDL

```
module M
{
  // definitions
};

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

C++11

```
namespace M
{
  // definitions
};

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



Basic Types

Remedy IT

Your challenge - our solution

IDL	C++11	Default value
short	int16_t	0
long	int32_t	0
long long	int64_t	0
unsigned short	uint16_t	0
unsigned long	uint32_t	0
unsigned long long	uint64_t	0
float	float	0.0
double	double	0.0
long double	long double	0.0
char	char	0
wchar	wchar_t	0
boolean	bool	false
octet	uint8_t	0



Constants

Remedy IT

Your challenge - our solution

IDL constants are mapped to C++11 constants using `constexpr` when possible

IDL

```
const string name = "testing";

interface A
{
    const float value = 6.23;
};
```

C++11

```
const std::string name {"testing"};

class A
{
public:
    static constexpr float value {6.23F};
};
```



String Types

Remedy IT

Your challenge - our solution

No need to introduce an IDL specific type mapping but leverage STL

IDL

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

C++11

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



Enumerations

Remedy IT

Your challenge - our solution

IDL enums map to C++11 strongly typed enums

IDL

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

C++11

```
enum class Color : uint32_t
{
    red,
    green,
    blue
};

Color mycolor {Color::red};
if (mycolor == Color::red)
{
    std::cout << "Correct color";
}
else
{
    std::cerr << "Incorrect color " <<
        mycolor << std::endl;
}
```



Sequence

Remedy IT

Your challenge - our solution

IDL unbounded sequence maps to `std::vector`

IDL

```
typedef sequence<long> LongSeq;  
  
typedef sequence<LongSeq, 3> LongSeqSeq;
```

C++11

```
typedef std::vector <int32_t> LongSeq;  
  
typedef std::vector <LongSeq> LongSeqSeq;  
  
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 << " ";  
}
```




Struct (1)

Remedy IT

Your challenge - our solution

IDL struct maps to a C++ class with copy and move constructors/assignment operators and accessors

IDL

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

C++11

```
class Variable  
{  
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

IDL struct maps to a C++ class with copy and move constructors/assignment operators and accessors

IDL

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

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;  
}
```



Array

Remedy IT

Your challenge - our solution

IDL array map to C++11 `std::array`

IDL

```
typedef long L[10];  
  
typedef string V[10];  
  
typedef string M[1][2][3];
```

C++11

```
typedef std::array <int32_t, 10> L;  
  
typedef std::array <std::string, 10> V;  
  
typedef std::array <std::array <std::array  
    <std::string, 3>, 2>, 1> M;  
  
// Initialize the array  
F f = { {1, 2, 3, 4, 5} }  
  
// Check the size of an array  
if (f.size () != 5)
```



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



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
```



Reference Types (3)

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 : public
    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 ();
```



Reference Types (4)

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)
```



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



IDL Traits

- For each IDL type a `IDL::traits<>` specialization will be provided
- The IDL traits contain a set of members with meta information for the specific IDL type
- The IDL traits are especially useful for template meta programming



Implement Interfaces

- 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`



CORBA AMI

Remedy IT

Your challenge - our solution

- TAOX11 has support for the callback CORBA AMI support
- The TAO AMI implementation has the disadvantage that when AMI is enabled for an IDL file all users have to include the TAO Messaging library
- TAOX11 separates CORBA AMI into a new set of source files, a client not needing AMI doesn't have to link any CORBA Messaging support!
- All `sendc_` operations are member of a derived CORBA AMI stub, not part of the regular synchronous stub



CORBA AMI Traits

- Instead of remembering some specific naming rules a new `CORBA::amic_traits<>` trait has been defined
- Contains the concrete types as members
 - `replyhandler_base_type`: the base type for implementing the reply handler servant
 - `replyhandler_servant_ref_type`: the type for a reference to the servant of the reply handler
 - `ref_type`: the client reference to the stub with all synchronous operations



CORBA AMI Example

Remedy IT

Your challenge - our solution

```
// Obtain a regular object reference from somewhere, Test::A has one method called foo
IDL::traits<Test::A>::ref_type stub = ...;

// Narrow the regular object reference to the CORBA AMI stub (assuming this has been
// enabled during code generation
CORBA::amic_traits<Test::A>::ref_type async_stub =
    CORBA::amic_traits<Test::A>::narrow (stub);

// Assume we have a Handler class as reply handler implemented, create it and
// register this as CORBA servant
CORBA::amic_traits<Test::A>::replyhandler_servant_ref_type h =
    CORBA::make_reference<Handler> ();
PortableServer::ObjectId id =
    root_poa->activate_object (h);
IDL::traits<CORBA::Object>::ref_type handler_ref =
    root_poa->id_to_reference (id);
CORBA::amic_traits<Test::A>::replyhandler_ref_type test_handler =
    CORBA::amic_traits<Test::A>::replyhandler_traits::narrow (handler_ref);

// Invoke an asynchronous operation, can only be done on async_stub, not on stub
async_stub->sendc_foo (test_handler, 12);

// But we can also invoke a synchronous call
async_stub->foo (12);
```



Valuetypes

- Valuetypes are mapped to a set of classes which are accessible through the `IDL::traits<>`
 - `IDL::traits<>::base_type` provides the abstract base class from which the valuetype implementation could be derived from
 - `IDL::traits<>::obv_type` provides the object by value class that implements already all state accessors and from which the valuetype implementation can be derived from
 - `IDL::traits<>::factory_type` provides base class for the valuetype factory implementation



Remedy IT

Your challenge - our solution

Example CORBA 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 ();
};
```




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<Test::Hello>::ref_type hello = IDL::traits<Test::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 CORBA::servant_traits<Test::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<Test::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<Test::Hello>::ref_type hello =
            IDL::traits<Test::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;
}
```



Auto specifier

- C++11 has support for auto as new type specifier
- The compiler will deduce the type of a variable automatically from its initializers
- Will simplify the CORBA example further



CORBA client

Remedy IT

Your challenge - our solution

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

        // Create the object reference
        auto obj = orb->string_to_object ("file://test.ior");

        // Narrow it to the needed type
        auto hello = IDL::traits<Test::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 CORBA::servant_traits<Test::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
        auto _orb = CORBA::ORB_init (argc, argv);

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

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

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

        // Put the IOR on disk
        auto 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



Remedy IT

Your challenge - our solution

Want to know more?

- Look at the TAOX11 website at <http://taox11.remedy.nl>
- Check the Remedy IT provided examples at <https://github.com/RemedyIT/idl2cpp11>
- Request your free-of-charge evaluation license, see <http://swsupport.remedy.nl>
- Contact us, see <http://www.remedy.nl>



Contact

Remedy IT

Your challenge - our solution

Remedy IT
Postbus 81
6930 AB Westervoort
The Netherlands

tel.: +31(0)88 053 0000

e-mail: sales@remedy.nl

website: www.remedy.nl

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

Slideshare: [RemedyIT](https://www.slideshare.net/RemedyIT)

Subscribe to our [mailing list](#)