



# Architecture of the CORBA Component Model

C++ Language Mapping: Data Types

# Requirements

- 🔦 Intuitive and easy to use.
- 🔦 Preserve commonly used C++ idioms, and “feel” like normal C++ as much as possible.
- 🔦 Should be type-safe.
- 🔦 Should be efficient in use of memory and CPU cycles.
- 🔦 Must be reentrant, to support multi-threading.
- 🔦 Must preserve location transparency.

Conflicting goals (efficiency vs. convenience). OMG favors efficiency:

- 🔦 It is possible to put a convenience layer on top of the standard mapping.
- 🔦 IDL is used for in-process interfaces, where any overhead of the mapping is expensive.
- Even though the mapping is large and complex, it is also consistent and type-safe.

# Identifiers

- ✦ Preserves names without change in C++

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

maps to

```
enum Color{ red, green, blue }; // C++
```

- ✦ Keywords are prefixed with `_cxx_` (should be avoided)
- ✦ Preserves scoping: `Outer::Inner` maps to `Outer::Inner`

# Modules

- ✦ Map to namespaces.

```
module Outer{  
    module Inner{  
        ...  
    };  
};
```

maps to

```
namespace Outer{  
    namespace Inner {  
        ...  
    }  
}
```

- ✦ C++ allows to omit namespace names after using directives:

```
using namespace Outer::Inner;
```

# The CORBA Namespace

- ✦ CORBA defines a standard IDL module CORBA, which maps to the C++ namespace CORBA.
- ✦ Primitive types of IDL also map to constructs in namespace CORBA
  - All primitive types except string/wstring map to a typedef in the CORBA namespace
  - A number of supporting functions in the CORBA namespace

# Primitive Types

IDL Type	C++ Type
<code>boolean</code>	<code>CORBA::Boolean</code>
<code>char</code>	<code>CORBA::Char</code>
<code>wchar</code>	<code>CORBA::WChar</code>
<code>short</code>	<code>CORBA::Short</code>
<code>long</code>	<code>CORBA::Long</code>
<code>long long</code>	<code>CORBA::LongLong</code>
<code>unsigned short</code>	<code>CORBA::UShort</code>
<code>unsigned long</code>	<code>CORBA::ULong</code>
<code>unsigned long long</code>	<code>CORBA::ULongLong</code>
<code>octet</code>	<code>CORBA::Octet</code>

## Primitive Types (2)

IDL Type	C++ Type
<code>float</code>	<code>CORBA::Float</code>
<code>double</code>	<code>CORBA::Double</code>
<code>long double</code>	<code>CORBA::LongDouble</code>
<code>string</code>	<code>char*</code>
<code>wstring</code>	<code>CORBA::WChar*</code>
<code>any</code>	<code>CORBA::Any</code>

# Overloading on Basic Types

- ✦ All basic types map to distinct C++ types, except:
  - char, boolean, octet map to the same 8-bit type
  - wchar maps to an integer type or `wchar_t`
- ✦ boolean may map to `bool`

# String and Wide String Mapping

- ✦ Strings are mapped to `char*`, wide strings to `CORBA::Wchar*`
- ✦ Bounds must be guaranteed by the application
- ✦ Helper functions to allocate and deallocate strings:

```
namespace CORBA{  
    char* string_alloc(ULong len);  
    char* string_dup(const char*);  
    void string_free(char*);  
    static WChar* wstring_alloc(ULong len);  
    static WChar* wstring_dup(const WChar*);  
    static void wstring_free(WChar*);  
}
```

## String and Wide String Mapping (2)

- ✦ Strings are terminated with `\0`; actual storage may be larger
- ✦ `new/delete` should not be used

# Mapping for Constants

- ✚ Global constants map to C++ file scope constants
- ✚ Constants defined in an interface map to static members of a C++ class

```
// IDL
```

```
const long MAX_ENTRIES=10;
```

```
interface NameList {
```

```
    const long MAX_NAMES = 20;
```

```
}
```

```
// C++
```

```
const CORBA::Long MAX_ENTRIES = 10;
```

```
class NameList {
```

```
    static const CORBA::Long MAX_NAMES = 20;
```

```
};
```

# Mapping for Enumerations

- ✦ IDL enumerations map to C++ enumerations
- ✦ dummy value may be generated to enforce 32-bit representation

```
// IDL
enum Color {red, green, blue, black, mauve,
            orange};

// C++
enum Color{
    red, green, blue, black, mauve, orange,
    _Color_dummy=0x80000000
};
```

# Variable-length types & `_var` Types

- ✦ Variable-length values must be dynamically allocated
- ✦ Returning variably-sized data in C++?
  - Return static memory?
  - Static pointer to dynamic memory?
  - Caller allocated memory?
  - Return pointer to dynamically-allocated memory?

# \_var Types: Motivation

- ✦ Memory management convention:
  - Producer allocates, consumer deallocates
  - Client side, **in** parameters: caller allocates and deallocates
  - Server side, **in** parameters: ORB (OA) allocates and deallocates
  - Client side, results: ORB allocates, caller deallocates
  - Server side, results: Servant allocates, OA deallocates
  - Client side, inout parameters: ...
  - ...
- ✦ Solution: `_var` types
  - Wrappers for lower-level mapped types
  - Manage storage for lower-level type
  - generated for both fixed-size and variable-size types

# Generated \_var types

IDL type	C++ type	Wrapper C++
string	char*	CORBA::String_var
any	CORBA::Any	CORBA::Any_var
interface foo	foo_ptr	class foo_var
struct foo	struct foo	class foo_var
union foo	class foo	class foo_var
typedef sequence<X> foo	class foo	class foo_var
typedef X foo[10]	typedef X foo[10]	class foo_var

# CORBA::String\_var

```
class String_var{
public:
    String_var();
    String_var(char*);
    String_var(const char*);
    String_var(const String_var&s);
    ~String_var();
    //...
private:
    char *s;
};
```

## CORBA::String\_var (2)

```
class String_var{
  //...
  String_var& operator=(char*p);
  String_var& operator=(const char*p);
  String_var& operator=(const String_var &s);
  operator char*();
  operator const char*()const;
  operator char* &();
  char& operator[](ULong index);
  char operator[](ULong index)const;
};
```

## CORBA::String\_var (3)

```
class String_var{  
  // ...  
  const char*      in() const;  
  char*&           inout();  
  char*&           out();  
  char*            _retn();  
};
```

# Assignment of String\_var

- ✦ Basic memory management functions

```
char *CORBA::string_alloc(int length);
```

```
char *CORBA::string_dup(const char*);
```

```
void CORBA::string_free(char*);
```

- ✦ operator=(const char\*) copies

```
String_var v;
```

```
v = "Hello, World";
```

- Pitfall: What if string literal is char\*?

- ✦ operator=(char\*) takes over memory ownership

```
v = CORBA::string_dup("Hello, World");
```

- ✦ operator=(const String\_var&) copies

# Assignment from String\_var

✦ Assignment invokes conversion:

```
const char *s1 = v;
```

```
char * s2 = v;
```

✦ Conversion returns pointer to internal representation

✦ Pointer may become invalid if underlying string is released

## Passing strings for read access

- ✦ Using `String_var` as a parameter type causes unnecessary copy:

```
void print_string(String_var v)
{
    cout << v << endl;
}

int main()
{
    String_var v = CORBA::string_dup("Hello");
    print_string(v);
}
```

- ✦ use `const char*` instead.

# Passing strings for update access

⚠ Using `String_var&` will cause temporaries if the actual argument is `char*`

⚠ Using `char*&` allows to pass either `String_var` or `char*`

```
void update_string(char* &s)
```

```
{
```

```
    CORBA::string_free(s);
```

```
    s = CORBA::string_dup("New string");
```

```
}
```

```
int main()
```

```
{
```

```
    CORBA::String_var v = CORBA::string_dup("Hello");
```

```
    update_string(v); // invokes operator char*&
```

```
}
```

# Returning strings

- ✦ `String_var::_retn()` retrieves embedded copies, and yields ownership
  - Using `String_var` may be needed in the presence of exceptions

```
char * get_line()
```

```
{
```

```
    CORBA::String_var v = CORBA::string_dup(buffer);
```

```
    if(db.close()) {
```

```
        throw DB_CloseFailure();
```

```
    }
```

```
    return v2._retn;
```

```
}
```

# Strings and iostreams

✦ operator<< is overloaded for CORBA::String\_var.

# Mapping for Structures

- ✂ Distinction between fixed-length and variable-length structures
- ✂ Fixed-length structures map to **struct**, with one element per field

```
// IDL
struct Details {
    double weight;
    unsigned long count;
};

// C++
class Details_var;
struct Details{
    CORBA::Double weight;
    CORBA::ULong count;
    typedef Details_var _var_type;
}
```

# Mapping for Variable-Length Structures

- ✦ Variable-length fields map to memory-managed class

```
// IDL
struct Fraction{
    double numeric;
    string alphabetic;
};

// C++
class Fraction_var;
struct Fraction{
    CORBA::Double numeric;
    CORBA::String_mgr alphabetic;
    typedef Fraction_var _var_type;
};
```



# Memory Management for Structures

- ✦ Assignment to variable-length fields involves automatic memory management
  - string fields start off as an empty string
- ✦ Deletion of structure deallocates storage for all fields.

# Mapping for Sequences

- ✦ Sequences map to vector-like types (typically implemented as templates)
- ✦ typedef names are used as class names

```
// IDL
```

```
typedef sequence<string> StrSeq;
```

```
// C++
```

```
class StrSeq_var;
```

```
class StrSeq{
```

```
    //...
```

```
};
```

# Sequence Constructors

```
StrSeq(); // empty sequence
StrSeq(CORBA::ULong max); // reserve elements
StrSeq(CORBA::ULong max, // provide array of initial elements
CORBA::ULong len,
char **data;
CORBA::Boolean release=0);
StrSeq(const StrSeq&); // copy constructor, deep copy
~StrSeq(); // destructor
```



# Sequence Internal Storage

const char\*\*  
char\*\*

```
get_buffer()const;  
get_buffer(CORBA::Boolean orphan=0);
```

static char\*\*  
static void

```
allocbuf(CORBA::ULong nelems);  
freebuf(char**data);
```

## Other Sequence Elements

```
StrSeq& operator=(const StrSeq&);           // copy assignment  
typedef StrSeq_var _var_type;
```

# Mapping for Unions

✘ Cannot map to C++ unions: no discriminator in C++

```
// IDL
```

```
union U switch(short){
```

```
case 1:                long long_mem;
```

```
case 2: case 3:        char char_mem;
```

```
default:               string string_mem;
```

```
};
```

```
// C++
```

```
class U_var;
```

```
class U{
```

```
    //...
```

```
};
```

# Union Constructors

```
U(); // indeterminate initialization
U(const U&); // copy constructor
~U(); // destructor
U& operator=(const U&); // copy assignment
```

# Union Discriminators

```
CORBA::Long  _d()const;  
void         _d(CORBA::ULong);
```

- ✚ Setting the discriminator is only allowed if it does not change the active member

# Union Members

- ✦ Setting a field activates it, and updates the discriminator

```
CORBA::Long long_mem()const;           // read access
void long_mem(CORBA::Long);           // write access
CORBA::Char char_mem()const;
void char_mem(CORBA::Char);
const char* string_mem()const;
void string_mem(char*);                // takes ownership
void string_mem(const char*);          // copies
void string_mem(const CORBA::String_var&); // copies
```

## Unions without a default case

```
// IDL
union AgeOpt switch(boolean){
    case TRUE:    unsigned short age;
};
```

✦ To activate the FALSE case, this union provides a `_default` method:

```
void _default();
```

# \_var-Classes for user-defined types

✦ Given an IDL type T, T\_var is generated

✦ Constructors:

```
T_var(); // initialize to NULL
```

```
T_var(T*); // assume ownership
```

```
T_var(const T&); // copy constructor
```

```
~T_var();
```

```
T_var& operator=(T*); // release current value, take over new one
```

```
T_var& operator=(const T_var&); // copy assignment
```

# T\_var Accessors

```
T*          operator->();          // treat as pointer, for member access
const T*    operator->()const;
T&          operator&();          // treat as object, for parameter passing
const operator T&()const;
T&          operator[](CORBA::ULong);    // for sequences
const T&    operator[](CORBA::ULong)const;
```

## T\_var example

```
// IDL
typedef sequence<string>    NameSeq;
// C++
NameSeq_var ns = new NameSeq;
ns->length(1);
ns[0] = CORBA::string_dup("Bjarne");

NameSeq_var ns2;
ns2 = ns; // deep copy
ns2[0] = CORBA::string_dup("Andrew"); // deallocates Bjarne, takes ownership of
Andrew
```

# Mapping for Other Types

✦ Not discussed here:

- fixed types
- arrays
- valuetypes