# IDL to C++ Mapping

# Language Mapping Requirements

- Intuitive and easy to use
- Natural for the target language
- Typesafe
- Memory and CPU efficient
- Work on architectures with segmented and limited memory
- Thread-safe
- Location-independent

# OMG IDL C++ Mapping

- Rather efficient than convenient
  - You can create more convenient mapping from the more efficient one, but not the other way around
  - Efficiency important for inter-process communication
- Advantages
  - Consistent
  - Typesafe
  - Easy to remember
- Header files are not very readable

# Mapping for Identifiers

- Unchanged in generated C++ code
  - `enum Color { red, green, blue };  → enum Color { red, green, blue };`
- Preserves the scope from z IDL
  - `Outer::Inner → Outer::Inner`
- Problems with C++ identifiers
  - `enum class { if, this };  → enum _cxx_class { _cxx_if, _cxx_this };`
- Avoid identifiers containing double underscore
  - `typedef long my__long → typedef long my__long`
- They are reserved for implementation in C++

# Mapping for Modules

```
module Outer {
    // …
    module Inner {
        // …
    };
};
```

→

```
namespace Outer {
    // …
    namespace Inner {
        // …
    }
}
```

- Mapped for namespace
- Allow for application of **`using`** directive
- Definition of types and interfaces in module **`CORBA`**
  - **`namespace CORBA`** contains appropriate definitions

# Mapping for Basic Types

| IDL | C++ |
|---|---|
| short | `CORBA::Short` |
| long | `CORBA::Long` |
| long long | `CORBA::LongLong` |
| unsigned short | `CORBA::UShort` |
| unsigned long | `CORBA::ULong` |
| unsigned long long | `CORBA::ULongLong` |
| float | `CORBA::Float` |
| double | `CORBA::Double` |
| long double | `CORBA::LongDouble` |
| char | `CORBA::Char` |
| wchar | `CORBA::WChar` |
| string | `char *` |
| wstring | `CORBA::WChar *` |
| boolean | `CORBA::Boolean` |
| octet | `CORBA::Octet` |
| any | `CORBA::Any` |

# Mapping for type `string`

- Memory management
  - Not **`operator new`**, **`operator delete`**, **`malloc()`**, **`free()`**
  - Special functions in **`namespace CORBA`**

```
namespace CORBA {
// ...
static char * string_alloc(ULong len);
static char * string_dup(const char *);
static void string_free(char*);
// similiarly wstring_alloc() etc.
};

char * p = CORBA::string_alloc(5); // allocates 6 bytes
strcpy(p, ”Hello”);

char* p = CORBA::string_dup(”Hello”);
```

# Mapping for Constants

- Global IDL constants mapped to C++ constants with the file scope, constants within the interface mapped for static constants inside class:

```
const long MAX_ENTRIES = 10;


interface NameList {

    const long MAX_NAMES = 20;

};
```

→

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


class NameList {

public:

  static const CORBA::Long MAX_NAMES = 20;

};
```

- String constants mapped to constant pointers to constants:
  - `const string MSG1 = “Hello”;` → `const char* const MSG1 = “Hello”;`

# Mapping for Enumeration Types

- IDL:

```
enum Color {red, green, blue, black, purple, orange };
```

- C++:

```
enum Color {
red, green, blue, black, purple, orange,
_Color_dummy=0x80000000 // Force 32-bit size
};
```

- The name of identifier forcing 32 bits is not specified
- The values of individual labels can be different in different programming languages

# Variable-length Types and `_var` Types

- The size of variable-length types is unknown at the compilation time and thus they require dynamic allocation
- Memory management convention:
  - Callee allocates memory
  - Caller frees memory
- Two levels of mapping for C++
  - Low-level - the programmer allocates and frees memory explicitly
  - Higher level - smart pointers – **`_var`** types. Automatic memory management
- Programmer can choose any of these levels

# Memory Management for Variable-length Types

- Variable-length types:
  - Strings and wide strings
  - Object references
  - The **`any`** type
  - Sequences
  - Structures and unions containing (recursively) variable-length types
  - Arrays with elements of variable-length types

| IDL Type | C++ Type | Wrapper C++ Type |
|---|---|---|
| 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 |

# The `String_var` Class

- The "smart pointer" class for text strings
- Constructors/destructor:
  - **`String_var()`** – initializes with NULL pointer
  - **`String_var(char*)`** – class becomes the owner of the pointer (is responsible for freeing it)
  - **`String_var(const char*)`** – class performs deep copy
  - **`String_var(const String_var&)`** - class performs deep copy
  - **`~String_var()`** - calls **`CORBA::string_free`**

# The String_var Class (contd.)

```
{
  CORBA::String_var s(”Hello”); // possible shallow copy
 // type of string literal in standard C++ is const char*,
 // but in older compilers it may be char *
 // ...
} // Oops... Destructor calls
  // string_free() on a pointer to a string constant.

{
  CORBA::String_var s(CORBA::string_dup(”Hello”));
  // ...
} // No memory leak here. Destructor calls
  // string_free().

const char * message = ”Hello”;
// ...
{
  CORBA::String_var s(message); //makes a deep copy
  // ...
} // destructor deallocates its own copy only
```

# The String_var Class (contd.)

- The assignment operators

```
String_var & operator=(char *);
String_var & operator=(const char *);
String_var & operator=(const String_var &);
```

- They operate like the corresponding constructors, they free memory before assignment

```
CORBA::String_var target;
target = CORBA::string_dup(″Hello″); //target takes ownership
CORBA::String_var source;
source = CORBA::string_dup(″World″); //source takes ownership
target = source; //Deallocates ″Hello″ and takes ownership
                 //of deep copy of ″World″
```

# The String_var Class (contd.)

- The conversion operators

```
operator char *()
operator const char *() const
```

- Allow to pass **String_var** to the function accepting **char * or const char ***

```
CORBA::String_var
s=CORBA::string_dup(”Hello”);
size_t len;
len = strlen(s);
```

- The conversion to reference to a pointer allows to pass string to the function with the header like **void update_string(char * &);**

```
operator char * &()
```

# The String_var Class (contd.)

- The indexing operators

```
char & operator[](ULong)
char operator[](Ulong) const
```

```
CORBA::String_var s =
CORBA::string_dup(”Hello”);
cout << s[4] << endl;
```

# The problems with String_var

- Assignment of String_var to a pointer

```
CORBA::String_var s1 = CORBA::string_dup(”Hello”);
const char * p1 = s1; //Shallow assignment
char* p2;
{
  CORBA::String_var s2 = CORBA::string_dup(”World”);
  p2 = s2; //Shallow assignment
  s1 = s2; //Deallocate ”Hello”, deep copy ”World”
}; //Destructor deallocates s2 (”World”)

cout << p1 << endl; // Whoops, p1 points nowhere
cout << p2 << endl; // Whoops, p2 points nowhere
```

# Passing Strings as Read-only Parameters

- Wrong:

```
void print_string(CORBA::String_var s)
{
  cout << ″String is \″″ << s << ″\″″ << endl;
}
int main()
{
  CORBA::String_var msg1 = CORBA::string_dup(″Hello″);
  print_string(msg1);
  char* text=″World″;
  print_string(text);//oops...
  return 0;
}
```

# Passing Strings as Read-only Parameters

- Right:

```
void print_string(const char* s)
{
  cout << ”String is \”” << s << ”\”” << endl;
}
int main()
{
  CORBA::String_var msg1 = CORBA::string_dup(”Hello”);
  print_string(msg1);
  char* text=”World”;
  print_string(text);
  return 0;
}
```

# Passing Strings as Read/Write Parameters

```
void update_string(char* &s)
{
  CORBA::string_free(s);
  s = CORBA::string_dup(″New string″);
}
int main()
{
  CORBA::String_var sv = CORBA::string_dup(″Hello″);
  update_string(sv);
  cout << sv << endl; //prints ″New string″
  char * p = CORBA::string_dup(″Hello″);
  update_string(p);
  cout << p << endl; //prints ″New string″
  CORBA::string_free(p);
  return 0;
}
```

# Explicit Conversion Functions

- `const char * in() const`
- `char * & inout()`
- `char * & out()`
  - dealocates current string before returning the reference
- `char * _retn()`
  - returns the pointer, is no longer the owner (will not try to free it)

# Mapping for Fixed-point Types

- Class Fixed
- Constructors:
  - `Fixed f  = 999;      // As if IDL type fixed<3,0>`
  - `Fixed f1 = 1000.0;   // As if IDL type fixed<4,0>`
  - `Fixed f2 = 1000.05;  // As if IDL type fixed<6,2>`
  - `Fixed f3 = 0.1;      // As if IDL type fixed<18,17>`
- Caution: 0.1 may not have the exact floating-point representation and be represented as 0.10000000000000001, thus the **`fixed<18,17>`** type. It is better to initialize these types of constants with text strings.
  - `Fixed f3 = ”0.1”     // As if IDL type fixed<2,1>`
  - `Fixed f4 = ”01.30D”  // As if IDL type fixed<2,1>`

# Mapping for Structures

- 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;
    // member functions here
};
```

- The member functions should be ignored, as they are implementation-dependent.
- **`_var_type`** is used by templates
- **`String_mgr`** behaves in the same way as **`String_var`** apart from its default constructor initializing the object with an empty string instead of the NULL pointer. The name **`String_mgr`** is implementation-dependent.

# Memory Management for Structures

- Automatic memory management inside the structure

```
{
Fraction f;
f.numeric = 1.0/3.0;
f.alphabetic = CORBA::string_dup(”one third”);
// ...
} //No memory leak here
```

- Dynamic allocation using (possibly overloaded) **new** and **delete**

```
Fraction * f = new Fraction();
// ...
delete f;
```

# Mapping for Sequences

- IDL:

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

- C++:

```
class StrSeq {
  ...
};
```

```
StrSeq mySeq; // Default constructor, length is zero.
mySeq.length(9); // Length is nine, default constructed strings.
mySeq.length(2); // Destroys last seven strings, length is two.
```

- Length() appends or erases elements at the end of the sequence

# Mapping for Sequences (contd.)

- Sequences have overloaded **`operator[]`**. Are indexed in range **`0 .. length-1`**.

```
StrSeq mySeq;    // Default constructor, length is zero.
mySeq.length(9); // Length is nine, default constructed strings.
mySeq[0] = CORBA::string_dup("Hello"); // Overwrites existing element.
cout << mySeq[0]; // Prints "Hello".
cout << mySeq[9]; // Undefined behavior. Core dump possible.
```

- Constructor with the maximum length hint

```
StrSeq(CORBA::Ulong max);
StrSeq mySeq(10); // 10 is maximum, but length is zero.
mySeq.length(20); // Maximum doesn't limit sequence length
```

  - Guarantees, that the elements will not be moved while length < maximum
  - Does not guarantee preallocation, allocation of contiguous area nor allocation of exacly 10 elements
- The following snippet contains an error. What error?

```
StrSeq mySeq(10);
for (CORBA::Ulong I=0; I<10; I++)
 mySeq[I] = CORBA::string_dup("Error Here");
```

# Mapping for Sequences (contd.)

- The constructor with the buffer argument

```
    StrSeq(CORBA::Ulong   max,
           CORBA::Ulong   len,
           char **        data,
           CORBA::Bolean  release = 0);
```

  - Allows to use a preallocated buffer
  - Can be used to transmit binary data as a sequence of octets
  - If **`release`**!=0, buffer must be allocated using **`StrSeq::allocbuf()`**, and will be freed with **`StrSeq::freebuf()`**.
- Bounded-length sequences
  - IDL: **`typedef sequence<string, 100> StrSeq;`**
  - C++: identical as for unlimited length, but:
    - The maximum is hardcoded
    - An attempt to increase length above this limit gives undefined results

# Limitations of Sequences

- Inserting elements in the middle of a sequence has a cost O(n)
    - Required copying of individual elements in order to make place for new elements in the middle

# Mapping for Arrays

- IDL arrays mapped to C++ arrays
- IDL:

```
typedef float FloatArray[4];
```

- C++:

```
typedef CORBA::Float FloatArray[4];
typedef CORBA::Float FloatArray_slice;
FloatArray_slice* FloatArray_alloc();
FloatArray_slice* FloatArray_dup(const FloatArray_slice*);
void FloatArray_copy(FloatArray_slice* to, FloatArray_slice* from);
void FloatArray_free(FloatArray_slice*);
```

- Dynamically allocated arrays must use functions **`_alloc()`** and **`_free()`**
- The user is responsible for memory management

```
FloatArray_slice* a = FloatArray_alloc(); // 4 elements.
A[0] = 123.4;
FloatArray_free(a);
```

- **`FloatArray_copy`** implements deep copying

```
FloatArray_copy(a,b); // copy b to a
```

# Mapping for Unions

- Mapped to classes in C++
- IDL:

```
union U switch (char) {
  case ’L’: long long_mem;
  case ’C’: char char_mem;
  default: string string_mem;
};
```

- Usage in C++:

```
U my_u; // Not initialized.
my_u.long_mem(99);          // Activate long_mem.
assert(my_u._d() == ’L’);   // Verify with accessor
assert(my_u.long_mem() == 99);
my_u.char_mem(’X’);         // Activate char_mem.
assert(my_u._d() == ‘C’);
```

- You cannot change the discriminator, if it would activate or deactivate member:

```
U my_u;          // Not initialized.
my_u._d(’L’);  // Undefined behavior.
```

# Unions without `default` Label

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

- Additional function **`_default()`** setting the active member to default

```
AgeOpt my_age;
my_age._default(); set discriminator to false

AgeOpt my_age;
my_age._d(0); //illegal
```

# Unions with Fields of Complex Types (Structure, Union, Sequence, `fixed`)

- Additional 3 functions:

```
struct Details {
  double weight;
  long count;
};
typedef sequence<string> TextSeq;
union ShippingInfo switch (long) {
case 0:
  Details packaging_info;
default:
  TextSeq other_info;
};
```

```
class ShippingInfo {
public:
//...
const Details & packaging_info() const;
    //Accessor
void packaging_info(const Details &);
    //Modifier
Details & packaging_info(); //Referent

const TextSeq & other_info() const;
    //Accessor
void other_info(const TextSeq &); //Modifier
TextSeq & other_info(); //Referent
};
```

# Unions Containing Anonymous Sequences

```
union Link switch (long) {
case 0:
  typeA ta;
case 1:
  typeB tb;
case 2:
  sequence<Link> sc;
};

class Link {
public:
typedef some_internal_identifier _sc_seq;
// ...
};
```

# The `_var` Classes

- The IDL compiler generates the **`_var`** classes for user-defined types (constant- and variable-length)

```
{
   StrSeq_var sv = new StrSeq;
   sv->length(3);
   …
} // ~StrSeq() deallocates sequence
```

- Overloaded **`operator->`**
- **`in()`**, **`out()`**, **`inout()`** and **`_retn()`** for parameter passing
- Assume dynamic memory allocation