IDL to C++ Mapping

Wymagania na Language Mapping

- Intuicyjny i łatwy w użyciu
- Naturalny dla języka docelowego
- Bezpieczny ze względu na typy
- Wydajny pamięciowo i obliczeniowo
- Działać na architekturach z segmentowaną i ograniczoną pamięcią
- Musi być wielobieżny
- Powinien być niezależny od lokalizacji

OMG IDL C++ Mapping

- Bardziej wydajny niż wygodny
  - można stworzyć mapping wygodniejszy ale mniej efektywny
  - wydajność istotna ze względu na komunikację wewnątrz procesu
- Zalety
  - spójny
  - bezpieczny ze względu na typy
  - łatwy do zapamiętania
- Pliki nagłówkowe są mało czytelne

Mapping dla identyfikatorów

- Bez zmian w wygenerowanym kodzie C++
  - **enum Color { red, green, blue };** → `enum Color { red, green, blue };`
- Zachowuje zasięgi z IDL
  - **Outer::Inner** → `Outer::Inner`
- Problemy z identyfikatorami w C++
  - `enum class { if, this };`
    → `enum _cxx_class { _cxx_if, _cxx_this };`
- Unikać identyfikatorów z podwójnym znakiem podkreślenie
  - `typedef long my__long` → `typedef long my__long`
  - w C++ zarezerwowane dla implementacji

Mapping dla modułów

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

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

- Mapowane na przestrzenie nazw
- Zezwala na stosowanie dyrektywy **`using`**
- Definicje typów i interfejsów w module **`CORBA`**
  - przestrzeń nazw **`CORBA`** zawiera odpowiednie definicje

Mapping dla typów podstawowych

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 dla typu string

- Zarzadzanie pamięcią
  - nie operator new, operator delete, malloc(), free()
  - specjalne funkcje w przestrzeni nazw **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 dla stałych

- Stałe globalne IDL mapowane na stałe C++ w zasięgu pliku, stałe wewnątrz interfejsu na statyczne stałe wewnątrz klasy

```
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;
};
```

- Stałe łańcuchowe mapowane jako stały wskaźnik do stałej

const string MSG1 = “Hello”; → `const char* const MSG1 = “Hello”;`

Mapping dla typów wyliczeniowych

- 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
  };
```
- Nazwa identyfikatora wymuszającego 32 bity nie określona w specyfikacji
- Wartości poszczególnych etykiet mogą być różne w różnych językach programowania

Typy zmiennej długości i typy `_var`

- Typy zmiennej długości mają rozmiary nieznane na etapie kompilacji i dlatego wymagają dynamicznej alokacji
- Zarządzanie pamięcią
  - funkcja wywoływana alokuje pamięć
  - funkcja wywołująca zwalnia pamięć
- Dwa poziomy mappingu dla C++
  - niskopoziomowy – programista musi zająć się alokacją i dealokacją pamięci
  - na wyższym poziomie – *smart pointers* – typy **`_var`**. Automatyczne zarządzanie pamięcią
- Programista może wybrać dowolny poziom

Zarządzanie pamięcią dla typów zmiennej długości

- Typy zmiennej długości
  - string i wide string
  - object reference
  - typ **`any`**
  - sekwencje
  - struktury i unie zawierające (rekursywnie) typy o zmiennej długości
  - tablice elementów o zmiennej długości

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

Klasa `String_var`

- Klasa „inteligentnego wskaźnika” dla łańcuchów tekstowych dla typu string.
- Konstruktory/destruktor:
  - **`String_var()`** – inicjalizuje zerowym wskażnikiem
  - **`String_var(char*)`** – klasa staje się właścicielem wskażnika (odpowiada za jego zwolnienie)
  - **`String_var(const char*)`** – klasa wykonuje głębokie kopiowanie
  - **`String_var(const String_var&)`** - klasa wykonuje głębokie kopiowanie
  - **`~String_var()`** - wywołuje **`CORBA::string_free`**

Klasa `String_var` (cd.)

```
{
  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
```

Klasa `String_var` (cd.)

- Operatory przypisania

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

- Działają analogicznie jak odpowiadające im konstruktory, przed przypisaniem zwalniają pamięć

```
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”
```

Klasa `String_var` (cd.)

- Operatory konwersji

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

- Umożliwiają przekazanie **`String_var`** do funkcji przyjmującej **`char *`** lub **`const char *`**

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

- Konwersja do referencji do wskaźnika umożliwia przekazanie łańcucha do funkcji o nagłówku typu **`void update_string(char * &);`**

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

Klasa `String_var` (cd.)

- Operatory indeksowania

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

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

Problemy ze `String_var`

- Przypisanie **`String_var`** do wskaźnika

```
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
```

Przekazywanie łańcuchów jako parametrów tylko do odczytu

- Źle

```
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;
};
```

Przekazywanie łańcuchów jako parametrów tylko do odczytu (cd.)

- Dobrze

```
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;
};
```

Przekazywanie łańcuchów jako parametrów do modyfikacji

```
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;
};
```

Funkcje jawnej konwersji

- `const char * in() const`
- `char * & inout()`
- `char * & out()`
  - dealokuje bieżący łańcuch przed zwróceniem referencji
- `char * _retn()`
  - zwraca wskaźnik, nie jest już jego właścicielem (nie będzie go zwalniać)

Mapping dla typów stałoprzecinkowych

- Klasa **`Fixed`**
- Konstruktory:

```
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>
```

- Uwaga: 0.1 może nie mieć dokładnej reprezentacji w formacie zmiennoprzcinkowym i być reprezentowana jako 0.10000000000000001, stąd typ **`fixed<18,17>`**. Dlatego najlepiej inicjalizować stałe tego typu łańcuchami tekstowymi.

```
Fixed f3 = "0.1"     // As if IDL type fixed<2,1>
Fixed f4 = "01.30D"     // As if IDL type fixed<2,1>
```

Mapping dla struktur

- 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
    };
```

- Funkcje składowe należy zignorować jako zależne od implementacji.
- **`_var_type`** jest używany przy korzystaniu z wzorców
- **`String_mgr`** zachowuje się jak **`String_var`** poza tym, że jego domyślny konstruktor inicjalizuje obiekt pustym łańcuchem zamiast zerowego wskaźnika. Nazwa **`String_mgr`** jest zależna od implementacji.

Zarządzanie pamięcią dla struktur

- Automatyczne zarządzanie pamięcią wewnątrz struktury

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

- Alokacja dynamiczna przy pomocy **`new`** i **`delete`** (być może przeciążonych)

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

Mapping dla sekwencji

- 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()`** dodaje albo kasuje elementy na końcu sekwencji

Mapping dla sekwencji (cd.)

- Sekwencje mają przeciążony **`operator[]`**. Są indeksowane w zakresie 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.
```

- Konstruktor ze wskazówką na maksymalną długość

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

  - Gwarantuje, że elementy nie będą przenoszone dopóki length < maksimum
  - Nie gwarantuje prealokacji, alokacji ciągłego obszaru ani dokładnie 10 elementów
  - Poniższy fragment zawiera błąd. Jaki?

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

Mapping dla sekwencji (cd.)

- Konstruktor z podaniem bufora

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

  - Pozwala na użycie prealokowanego bufora
  - Może być wykorzystany do transmisji danych binarnych jako sekwencji oktetów
  - Jeżeli release!=0, to bufor musi zostać zaalokowany przy pomocy **`StrSeq::allocbuf()`**, a zostanie zwolniony przy pomocy **`StrSeq::freebuf()`**.
- Sekwencje z ograniczoną długością
  - IDL: **`typedef sequence<string, 100> StrSeq;`**
  - C++: identycznie jak dla nieograniczonej długości, ale:
    - maksimum jest na stałe wpisane w kod źródłowy
    - próba zwiększania długości poza to maksimum daje niezdefiniowane efekty

Ograniczenia sekwencji

- Wstawianie elementów w środek sekwencji ma koszt O(n)
  - wymagane kopiowanie pojedynczych elementów na koniec w celu zrobienia miejsca na nowe elementy w środku

Mapping dla tablic

- Tablice IDL mapowane na tablice C++
- 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*);
```

- Tablice alokowane dynamicznie muszą używać funkcji **`_alloc()`** i **`_free()`**
- Użytkownik odpowiedzialny za zarządzanie pamięcią

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

- FloatArray_copy implementuje głębokie przypisanie

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

Mapping dla unii

- Mapowane do klas w C++
- IDL:

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

- Użycie w 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');
```

- Nie można zmienić dyskryminatora, jeżeli musiałoby to aktywować lub deaktywować składową:

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

Unie bez etykiety `default`

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

- Dodatkowa funkcja _default() ustawiająca aktywną składową na domyślną

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

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

Unie z polami złożonych typów (struktura, unia, sekwencja, `fixed`)

- Dodatkowe 3 funkcje

```
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
};
```

Unie zawierajace anonimowe sekwencje

```
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;
// ...
};
```

Klasy `_var`

- Kompilator IDL generuje klasy _var dla typów zdefiniowanych przez użytkownika (stałej i zmiennej długości)

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

- Przeciążony **`operator->`**
- **`in()`**, **`out()`**, **`inout()`** i **`_retn()`** dla przekazywania parametrów
- Zakładają dynamiczną alokację pamięci