Mapeamento de IDL para C++

Mapeamento dos tipos de dados básicos

Os tipos de dados básicos são mapeados conforme a tabela abaixo. Note que o mapeamento do tipo OMG IDL boolean define somente os valores 1 (TRUE) e 0 (FALSE); outros valores produzem comportamento indefinido.

OMG IDL	**C++**
short	CORBA::Short
long	CORBA::Long
unsigned short	CORBA::UShort
unsigned long	CORBA::ULong
float	CORBA::Float
double	CORBA::Double
char	CORBA::Char
boolean	CORBA::Boolean
octet	CORBA::Octet

Mapeamento de OMG IDL para C++

Cada tipo básico OMG IDL é mapeado em um typedef no módulo CORBA. Isto se deve ao fato de que alguns tipos, como short e long, podem ter diferentes representações em diferentes plataformas e as definições CORBA refletirão a representação apropriada. Por exemplo, em uma máquina de 64 bits em que um inteiro long é de 64 bits a definição de CORBA::Long ainda referiria a um inteiro de 32 bits.

Com exceção do boolean, char, and octet, o mapeamento dos tipos básicos deve ser distinguível um do outro com o propósito de sobrecarga (overloading). Isto é, pode-se com segurança escrever funções C++ de sobrecarga para Short, UShort, Long, ULong, Float, and Double.

Os programadores preocupados com portabilidade devem usar os tipos CORBA. Entretanto, alguns podem considerar que o uso destes tipos com a qualificação CORBA impacta a legibilidade. Em plataformas em que os tipos de dados C++ são idênticos aos tipos OMG IDL uma implementação em conformidade com a especificação pode portanto gerar os tipos nativos C++.

Para o tipo Boolean somente os valores 1 (representando TRUE - verdadeiro) e 0 (representando FALSE - falso) são definidos; outros valores produzem um comportamento indefinido. Uma vez numerosos pacotes de software e bibliotecas já definem sua próprias definições macro de TRUE e FALSE, este mapeamento não requer que estas definições sejam supridas pela implementação. A requisição de definições para TRUE e FALSE poderia causar problemas de compilação para as aplicações CORBA que fizessem uso daqueles pacotes ou bibliotecas. Ao contrário, recomenda-se que as aplicações em conformidade com a especificação simplesmente usem os valores 1 e 0 diretamente. Alternativamente para aqueles compiladores que dão suporte ao tipo novo bool pode-se

usar true e false.

Usando Strings

As strings sempre foram uma fonte de problemas. O padrão CORBA adota um mapeamento para strings na linguagem C++ que não é necessariamente intuitivo. A descrição seguinte foi parcialmente retirada da especificação CORBA.

O tipo string OMG IDL, de tamanho limitado ou não, é mapeado em um char* na linguagem C++. Os dados do tipo string são terminados com um NULL. O módulo CORBA define ainda uma classe String_var que contém um char* e automaticamente libera o ponteiro quando um objeto String_var é desalocado. Quando uma string_var é construída ou atribuída a partir de um char*, o char* é consumido e o acesso a string não pode mais ser feito através do char* original. Atribuição ou construção a partir de um const char* ou de outra String_var provoca uma cópia. A classe String_var fornece ainda algumas operações para conversão de e para valores char* e para o acesso aos caracteres de uma string. A descrição completa da interface String_var é dada no apêndice da especificação CORBA. Para strings alocadas dinamicamente, programas em conformidade com a especificação devem usar as seguintes funções do espaço de nomes CORBA:

```
// C++
namespace CORBA {
char *string_alloc( ULong len );
char *string_dup( const char* );
void string_free( char * );
...
}
```

A função string_alloc aloca dinamicamente uma string ou retorna um ponteiro null se a alocação não pode ser feita. Ela aloca len+1 caracteres de tal maneira que a string resultante tenha espaço suficiente para acomodar o caracter NULL de terminação. A função string_dup aloca dinamicamente espaço suficiente para acomodar uma cópia da string fornecida como argumento, incluindo o caracter NULL, copia a string e retorna um ponteiro para a nova string. Se a alocação falha um ponteiro NULL é retornado. A função string_free desaloca uma string que foi anteriormente alocada com string_alloc ou string_dup. A passagem de um ponteiro NULL para string_free é aceitável e o resultado é que nenhuma ação é executada.

Note que um array estático de char em C++ decai em um char*, portanto deve-se tomar muito cuidado ao atribuir um destes a uma String_var uma vez que a String_var assumirá que o ponteiro aponta para dados alocados via string_alloc e tentará desalocá-lo via string_free:

```
// C++
// A seguir existe um erro, uma vez que o char* deve apontar para
// dados alocados via string_alloc para que possam ser consumidos
String_var s = "string estatica"; // erro

// A seguir está certo uma vez que const char* é copiado
// não consumido
const char* sp = "string estatica";
s = sp;
s0 = (const char*)"outra string estatica";
```

Valores sem tipos

A manipulação de valores sem tipo é uma das forças da especificação CORBA. A classe C++ predefinida Any do espaço de nomes CORBA fornece este suporte. Uma instância da classe Any representa um valor de um tipo IDL qualquer. A classe Any define para cada tipo os operadores sobrecarregados >>= e <<=. Estes dois operadores são responsáveis pela inserção e extração de valores de dados. A fragmento de código a seguir demonstra o uso destes operadores:

```
// C++
CORBA::Any a;
// Inserção em any
a <<= (CORBA::ULong) 10;
// Extração de any
CORBA::ULong l;
a >>= l;
```

Ao final deste exemplo a variável l deve ter o valor 10. A biblioteca MICO (uma das implementações disponíveis da especificação CORBA) fornece definições destes operadores para todos os tipos básicos. Alguns destes tipos de dados são ambíguos no sentido que eles colidem com outros tipos básicos. Isto ocorre para os tipos IDL boolean, octet, char e string. Para cada um destes tipos CORBA prescreve um par de funções de suporte que ajudam a retirar esta ambigüidade. Para o tipo boolean por exemplo o uso destas funções é:

```
CORBA::Any a;
// Inserção em any
a <<= CORBA::Any::from_boolean( TRUE );
// Extração de any
CORBA::Boolean b;
a >>= CORBA::Any::to_boolean( b );
```

O uso das outras funções de suporte para octet, char e string é equivalente. Para as strings limitadas (de tamanho limitado) as funções de suporte from_string e to_string aceitam um parâmetro adicional long que reflete o limite.

Para cada tipo definido na especificação IDL o compilador IDL gera uma versão sobrecarregada dos operadores >>= e <<=. Por exemplo, dada a seguinte especificação IDL:

```
// IDL
struct S1 {
long x;
char c;
};
struct S2 {
string str;
};
```

O compilador IDL do MICO gerará automaticamente as definições de >>= e <<= para os tipos IDL S1 e S2. O fragmento de código a seguir demonstra o uso destes operadores:

```
1: void show_any( const CORBA::Any& a )
2: {
3: S1 s1;
4: S2 s2;
5:
6: if( a >>= s1 ) {
7: cout << "Encontrei struct S1" << endl;
```

```
8: cout << s1.x << endl;
9: cout << s1.c << endl;
10: }
11: if( a >>= s2 ) {
12: cout << "Encontrei struct S2" << endl;
13: cout << s2.str << endl;
14: }
15: }
16:
17: int main( int argc, char *argv[] )
18: {
19: //...
20: CORBA::Any a;
21:
22: S2 s2;
23: s2.str = (const char *) "Ola!";
24: a <<= s2;
25: show_any( a );
26:
27: S1 s1;
28: s1.x = 42;
29: s1.c = 'C';
30: a <<= s1;
31: show_any( a );
32: }
```

O programa main primeiro inicializa uma instância de S2 (linhas 22-24) e então chama a função show_any. A função show_any tenta extrair o valor contido em any. Este exemplo também demonstra como determinar se a extração foi bem sucedida ou não. O operador >>= retorna true (verdadeiro) se o tipo do valor contido no any casa com o tipo da variável do lado direito de >>=. Se o any contiver algo que não S1 e S2 então show_any falhará nos comandos if das linhas 6 e 11. O código completo dos exemplos acima está disponível no diretório mico/test/idl/14 da instalação do MICO.

Para alguns tipos IDL dois diferentes operadores >>= e <<= são fornecidos: uma versão que faz cópia e outra que não faz. A versão do operador <<= que faz cópia pega a referência para o tipo IDL e insere uma cópia desta em Any. A versão que não faz cópia pega o ponteiro para o tipo IDL e o move para Any sem fazer cópia. O usuário não deve acessar o valor inserido posteriormente. A versão do operador >>= que faz cópia pega a referência para o tipo IDL e copia o valor de Any neste. A versão que não faz cópia pega a referência do ponteiro para o tipo IDL e o aponta para o valor em Any. O usuário não deve liberar o valor retornado. A seguir alguns exemplos:

```
// IDL
struct foo {
long l;
short s;
};
// C++
CORBA::Any a;
// <<= com copia
foo f;
a <<= f;
// << sem copia
foo *f = new foo;
a <<= f;
// nao acesse 'f' aqui ...
// >>= com copia
foo f;
a >>= f;
```

```
// >>= sem copia
foo *f;
a >>= f;
// nao libere 'f'
// alterando 'a' invalida 'f'
```

Tipos construídos desconhecidos

A implementação MICO do Any oferece uma interface estendida para inserção e extração de tipos construídos que não são conhecidos em tempo de compilação. Esta interface também é usada pelos operadores <<= e >>= gerados pelo compilador IDL para os tipos construídos. Vamos dar uma olhada nos operadores gera dois para uma estrutura simples:

```
1: // IDL
2: struct foo {
3: long l;
4: short s;
5: };
6:
7: // C++
8: void operator<<= ( CORBA::Any &a, const foo &s )
9: {
10: a.type( _tc_foo );
11: a.struct_put_begin();
12: a <<= s.l;
13: a <<= s.s;
14: a.struct_put_end();
15: }
16:
17: CORBA::Boolean operator>>=( const CORBA::Any &a, foo &s )
18: {
19: return a.struct_get_begin() &&
20: (a >>= s.l) &&
21: (a >>= s.s) &&
22: a.struct_get_end();
23: }
```

Na linha 10 o operador <<= diz ao Any o TypeCode (_tc_foo) que deve ser inserido na estrutura. As constantes _tc_* são gerados pelo compilador IDL. Se você deseja inserir um tipo construído que não é conhecido em tempo de compilação você deve obter o TypeCode de algum lugar (por exemplo do repositório de interface) ou você terá que criar um usando os métodos ORB create_*_tc().

Na linha 11 o operador <<= abre a estrutura, insere os elementos a estrutura nas linha 12 e 13 e fecha a estrutura na linha 14. Enquanto isto o Any verifica se os itens inseridos estão corretos usando o TypeCode. Se um erro for detectado (por exemplo, o TypeCode diz que o primeiro elemento da estrutura é um short e você insere um float) o método correspondente ou o operador <<= retornará um FALSE. Se a estrutura contém um outro tipo construído você deverá fazer chamadas aninhadas de truct_put_begin() e struct_put_end() ou os métodos correspondentes para unions, exceptions, arrays, ou sequences.

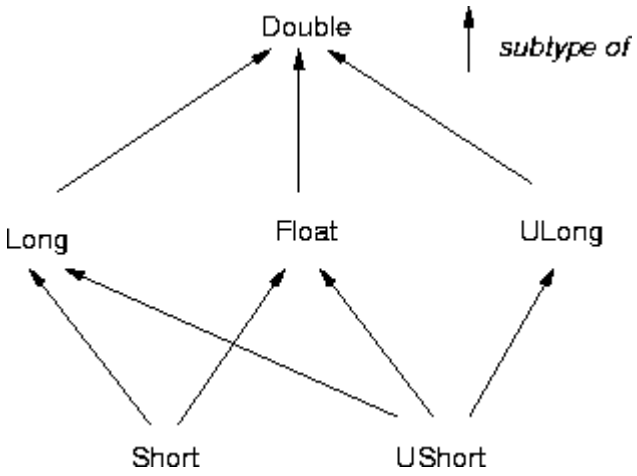
O operador >>= nas linhas 17-23 tem a mesma estrutura do operador <<= mas usa o operador >>= para extrair os elementos da estrutura e struct_get_begin() e struct_get_end() para abrir e fechar a estrutura. não é necessário especificar um TypeCode antes da extração pois o Any já o conhece.

Subtipos

Outra característica da implementação Any do MICO é o suporte a subtipos. Os operadores de extração do tipo Any implementam as regras de subtipos para tipos recursivos como prescrito pelo Modelo de Referência do Processamento Distribuído Aberto (Reference Model for Open Distributed Processing) ou RM-ODP. A idéia por trás da subtipagem é a seguinte: Imagine que você quer chamar um método CORBA

void bar (in long x);

mas quer passar um short como argumento ao invés do long requerido. Isto funciona na teoria uma vez que possivelmente o valor short é também um valor long o que significa que short é um subtipo de long. Falando mais genericamente um tipo T1 é um subtipo do tipo T2 se você pode passar T1 como um parâmetro nos casos em que T2 é esperado. Isto significa para os tipos básicos como o long que um tipo básico T1 é um subtipo do tipo T2 se o conjunto de valores possíveis de T1 é um subconjunto do conjunto de valores possíveis de T2. A figura abaixo mostra as relações de subtipos entre os tipos básicos CORBA. Em C++ o compilador pode automaticamente converter os tipos seguindo as setas, mas em uma aplicação distribuída isto não pode ser feito pelo compilador sozinho pois a ligação entre cliente e servidor é realizada em tempo de execução utilizando um trader ou serviço de nomes.



Relação de subtipos entre tipos básicos CORBA.

No caso da implementação MICO o tipo Any faz a verificação de subtipos em tempo de execução. Por exemplo:

```
// C++
CORBA::Any a;
a <<= (CORBA::Short) 42;
...
CORBA::Double d;
a >>= d;
```

vai funcionar pois short é um subtipo de double de acordo com afigura acima mas:

```
// C++
CORBA::Any a;
a <<= (CORBA::Long) 42;
...
CORBA::ULong d;
```

```
a >>= d;
```

vai falhar pois long não é um subtipo de unsigned long. Existe uma regra especial de subtipagem para tipos estruturados: Um tipo struct T1 é um subtipo de um tipo struct T2 se os elementos de T2 são supertipos dos primeiros elementos de T1. Por exemplo, a struct S1 é um subtipo da struct S2:

```
struct S1 {
short s;
long l;
};
struct S2 {
long s;
};
```

Ou seja, você pode colocar uma struct S1 em um Any e depois retirá-la como uma struct S2:

```
// C++
CORBA::Any a;
S1 s1 = { 10, 20 };
a <<= s1;
...
S2 s2;
a >>= s2;
```

Há regras similares para os outros tipos construídos.

Arrays e Unions

Um array é manipulado de uma maneira desajeitada em CORBA. O mapeamento C++ para a declaração de um array é clara. As coisas vão ficando um pouco mais complicadas quando os arrays são passados como parâmetros das operações. Os arrays são mapeados na definição de array C++ correspondente que permite a definição de dados estaticamente inicializados usando o array. Se o elemento do array é uma string ou uma referência de objeto, então o mapeamento usa os mesmos tipos dos membros de estruturas. Ou seja, uma atribuição de um elemento de um array vai liberar o valor antigo armazenado.

```
  // IDL
  typedef string V[10];
  typedef string M[1][2][3];
  // C++
  V v1; V_var v2;
  M m1; M_var m2;
  v1[1] = v2[1]; // libera o valor antigo, copia
  m1[0][1][2] = m2[0][1][2]; // libera o valor antigo, copia
```

No exemplo acima, as duas atribuições resultam na liberação automática dos valores antigos associados ao lado esquerdo da atribuição antes que o valor do lado direito seja copiado.

Como os arrays são mapeados em arrays C++ regulares, eles apresentam problemas especiais para o mapeamento do tipo Any. Para facilitar seu uso com o tipo Any MICO também fornece para cada array um tipo C++ distinto cujo nome consiste do nome do array seguido do sufixo `_forany`. Como os tipos `Array_var`, os tipos `Array_forany` permitem o acesso ao array associado. A interface do tipo `Array_forany` é idêntica a do tipo `Array_var`.

```
  // IDL
  typedef string V[10];
  // C++
  V_forany v1, v2;
  v1[0] = ...;  // Inicializa o array
  CORBA::Any any;
  any <<= v1;
  any >>= v2;   // v1 e v2 agora possuem valor idêntico
```

Além do mapeamento para o tipo `Array_forany` o padrão CORBA também define um mapeamento para um *array slice* (pedaço de array). Um array slice é um array com todas as dimensões originais menos a primeira. Os parâmetros de saída e resultados são manipulados via ponteiros para array slice. O array slice é nomeado como o array mais o sufixo `_slice`. Para a declaração do tipo M no exemplo acima, o compilador IDL vai gerar a seguinte definição:

```
  // Gerado pelo compilador IDL, C++
  typedef M M_slice[2][3];
```

Vamos considerar a seguinte especificação IDL (veja também `mico/test/idl/18`):

```
  // IDL
  // Note: long_arr e um array de tipo de dados de tamanho fixo
  typedef long long_arr[ 10 ];
  // Note: SS e um array de tipo de dados de tamanho variavel
  typedef string SS[ 5 ][ 4 ];
  interface foo {
    SS bar( in SS x, inout SS y, out SS z, out long_arr w );
  };
```

A implementação da interface `foo` será algo semelhante a:

```
class foo_impl : virtual public foo_skel
{
  //...
  SS_slice* bar( const SS ss1, SS ss2, SS_slice*& ss3, long_arr arr )
  {
    //...
    ss3 = SS_alloc();
    SS_slice *res = SS_alloc();
    return res;
  };
};
```

Note que o valor de retorno da operação `bar` é um ponteiro para array slice. Os parâmetros de saída quando o tipo é um array de tipos de dados de tamanho variável são manipulados via uma referência para ponteiro de um array slice. Para facilitar a gerência de memória com array slices o padrão CORBA prescreve o uso de funções especiais definidas no mesmo escopo que o tipo array. Para o array `SS`, as seguintes funções estarão disponíveis para um programa:

```
  // C++
  SS_slice *SS_alloc();
  SS_slice *SS_dup( const SS_slice* );
  void SS_free( SS_slice * );
```

A função `SS_alloc` aloca dinamicamente um array ou retorna um ponteiro null se a locação não puder

ser realizada. A função `SS_dup` aloca dinamicamente um novo array do mesmo tamanho que o array passado como argumento, copia cada elemento do argumento no novo array e retorna o ponteiro para o novo array. Se a alocação falhar um ponteiro null é retornado. A função `SS_free` desaloca um array que foi anteriormente alocado através das funções `SS_alloc` ou `SS_dup`. A passagem de um ponteiro null para `SS_free` é aceitável mas nenhuma ação é executada.

As unions e structs em CORBA-IDL permitem a definição de tipos de dados construídos. Cada um deles é definido como um conjunto de membros. Se uma estrutura é usada como parâmetro de entrada de uma operação, todos seus membros serão transmitidos enquanto no caso de uma union somente um dos seus membros será transmitido. O propósito de uma union IDL é similar a union C: reduzir o uso de memória. Isto é especialmente importante em um plataforma middleware em que menos espaço de memória significa menos dados a serem transmitidos pela rede. Deve-se considerar com carinho quando usar structs ou unions.

Um problema especial surge com unions quando elas são usadas como parâmetros de operações: como o objeto que as recebe sabe qual dos diferentes membros contém um valor válido? Para fazer a distinção neste caso, a union IDL é uma combinação de uma union c e comando switch. Cada membro é rotulado com um valor de um tipo discriminador (veja também `mico/test/idl/21`):

// IDL

```
typedef octet Bytes[64];
struct S { long len; };
interface A;
union U switch (long) {
  case 1: long x;
  case 2: Bytes y;
  case 3: string z;
  case 4:
  case 5: S w;
  default: A obj;
};
```

Na union U mostrada acima `long` é o tipo discriminador. Os valores de cada case devem pertencer a este tipo discriminador. Todos os tipos inteiros e enums são tipos discriminadores válidos. As unions são mapeadas em classes C++ com funções de acesso aos membros da union e ao discriminante. O construtor default da union não executa nenhuma inicialização visível a aplicação. É, portanto, um erro se a aplicação acessar a union antes de dar valor a ela. O construtor copia e o operador de atribuição executam uma cópia de dos parâmetros, sendo que o operador de atribuição libera o antigo armazenamento se necessário. O destrutor libera todos os armazenamentos de propriedade da union. O exemplo a seguir ajuda ilustra o mapeamento para o tipo union para a union U apresentada acima:

```
// codigo C++ gerado
typedef CORBA::Octet Bytes[64];
typedef CORBA::Octet Bytes_slice;
template<...> Bytes_forany;
struct S { CORBA::Long len; };
typedef ... A_ptr;
class U {
  public:
    //...
    void _d( CORBA::Long );
    CORBA::Long _d() const;
    void x( CORBA::Long );
```

```
    CORBA::Long x() const;
    void y( Bytes );
    Bytes_slice *y() const;
    void z( char* );             // libera armazenamento anterior, sem copia
    void z( const char* );       // libera armazenamento anterior, copia
    void z( const String_var& ); // libera armazenamento anterior, copia
    const char *z() const;
    void w( const S & ); // copia
    const S &w() const;  // acesso somente leitura
    S &w();              // acesso somente de escrita
    void obj( A_ptr ); // libera a refere a referencia de objeto antiga, duplica
    A_ptr obj() const; // não duplica
};
```

As funções de acesso ao discriminante da union possuem o nome `_d` para ser breve e evitar conflitos de nome. A função modificadora do discriminador `_d` só pode ser usada para definir o valor do discriminante para um valor dentro do mesmo membro da union. Além do `_d`, uma union com um membro default implícito fornece uma função `_default()` que define o valor do discriminante para um valor default legal. Uma union possui um membro default implícito se não tem um case default e nem todos os valores permitidos para o discriminante são listados.

A definição dos valores da union com as funções de acesso automaticamente define o discriminante e pode liberar o armazenamento associado aos valores prévios. A tentativa de obter um valor com as funções de acesso que não casa com nenhum discriminante resulta um comportamento indefinido. Se uma função de acesso de um membro da union com múltiplos valores de discriminantes válidos é usada para definir o valor do discriminante a implementação da union escolherá o valor do primeiro case na union (por exemplo, valor 4 para o membro `w` da union `U`), muito embora pudesse seu outro valor para aquele membro.

As restrições para o uso da função modificadora do discriminador `_d` são mostradas pelos exemplos baseados na definição da union `U` mostrada acima:

```
// C++
S s = ...;
A_ptr a = ...;
U u;
u.w( s );   // membro w selecionado, discriminador == 4
u._d( 4 );  // OK, membro w selecionado
u._d( 5 );  // OK, membro w selecionado
u._d( 1 );  // erro, membro diferente selecionado
u.obj( a ); // membro obj selecionado
u._d( 7 );  // OK, membro obj selecionado
u._d( 1 );  // erro, membro diferente selecionado
```

Como mostrado aqui, a função modificadora do `_d` não pode ser usada para chavear implicitamente entre membros diferentes da union. O exemplo seguinte mostra como a função membro `_default()`é usada:

```
//  IDL
  union Z switch(boolean) {
    case TRUE: short s;
  };
  // C++
  Z z;
  z._default();  // membro default implícito selecionado
  CORBA::Boolean disc = z._d(); // disc == FALSE
```

```
U u;           // union U do exemplo anterior
u._default();  // erro, _default() nao fornecido
```

Para a union `Z`, chamando a função membro `_default()` faz com que o os valores da union sejam compostos somente pelo valor discriminador `FALSE`, uma vez que não existe membro default explícito. Para a union `U`, chamando `_default()` provoca um erro de compilação pois `U` não possui uma declaração explícita do case default e portanto não possui a função membro `_default()`. Uma função membro `_default()` sé é gerada para union com membros defaults implícitos.

Para um array membro de union o `_d` retorna um ponteiro para array slice permitindo o acesso de leitura e escrita aos membros array. Para membros de tipo de array anônimo são gerados typedefs para o array diretamente na union. Por exemplo:

```
// IDL
union U switch (long) {
  case 1: long array[ 3 ][ 4 ];
};
// Codigo C++ gerado
class U {
  public:
  // ...
  typedef long _array_slice[ 4 ];
  void array( long arg[ 3 ][ 4 ] );
  _array_slice* array();
};
```

O nome do typedef do array slice de suporte é criado acrescentando-se antes do nome do membro da union um sublinhado e `_slice` ao final. No exemplo acima, o nome do membro array `_array` resulta em um typedef chamado `_array_slice` aninhado na classe union.

Módulos

Ao contrário de outras plataformas middleware CORBA não atribui um identificador único universal a uma interface. Para evitar choque de nomes CORBA oferece um espaço de nomes estruturado similar a estrutura de diretórios de um sistema de arquivo UNIX. Em IDL o escopo é definido pela palavra-chave `module`. Por exemplo, o fragmento de código IDL seguinte define dois módulos denominados `Mod1` e `Mod2` no mesmo nível:

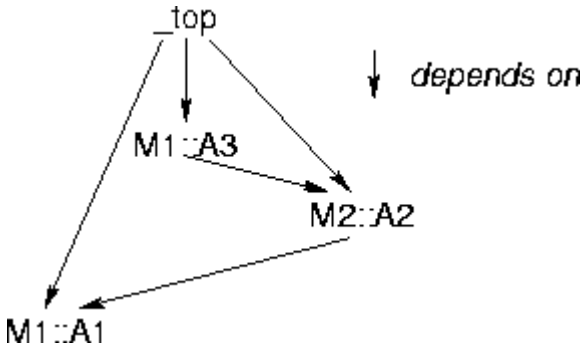
```
module Mod1 {
  //...
  interface foo;
};
module Mod2
{
  //...
};
```

declarações de módulo podem ser aninhadas o que leva a hierarquia de nomes mencionada acima. O mapeamento IDL para C++ oferece diferentes alternativas para mapear um módulo em C++. No caso dos compiladores C++ que dão suporte as características de espaço de nomes da linguagem C++ os módulos IDL são diretamente mapeados em namespaces C++. Infelizmente o compilador GNU atualmente não dá suporte a namespaces. Neste caso a especificação CORBA oferece duas alternativas: ou alterar o nome de tal maneira que o nome reflita o nome absoluto do identificador IDL separando os nomes por sublinhado (ex.: `Mod1_foo`) ou mapear o módulo IDL em uma struct C++.

A segunda alternativa tem duas desvantagens: sem um suporte apropriado para namespaces todos os nomes devem ser referenciados por seus nomes absolutos, isto é, não existe a palavra chave C++ `using` (note que isto também é verdade para a primeira alternativa). A segunda desvantagem tem relação com a possibilidade de reabrir módulos CORBA o que permite definições cíclicas:

```
module M1 {
  typedef char A;
};
module M2
{
  typedef M1::A B;
};
module M1 {  // reabre modulo M1
{
  typedef M2::B C;
};
```

A declaração de uma `struct` C++ deve ocorrer em uma localidade, isto é, a `struct` não pode ser reaberta. Portanto, mapear módulos IDL em structs C++ implica que a reabertura de módulos não pode se traduzida para C++. Contudo, se o compilador C++ der suporte a namespaces, o compilador IDL MICO permite a reabertura de módulos. O compilador IDL gera um grafo de dependência para computar a ordenação correta das definições IDL. A figura abaixo mostra o grafo de dependência para a especificação IDL acima.



Grafo de dependência.

Algumas vezes é necessário ter algum controle sobre os módulos de nível mais lato. Isto é usado, por exemplo, em `CORBA.h` onde algumas definições devem ser lidas uma por vez. O compilador IDL insere alguns `#define` no arquivo `.h` gerado. A definição e retirada das definições permite ler uma definição de módulo por vez. sendo dados dois módulos `Mod1` e `Mod2` como acima, o fragmento de código C++ seguinte demonstra como isto é feito:

```
1:  // Este #includes precisam ser feitos manualmente se
2:  // MICO_NO_TOPLEVEL_MODULES é definido
3:  #include <CORBA.h>
4:  #include <mico/template_impl.h>
5:
```

```
 6:  #define MICO_NO_TOPLEVEL_MODULES
 7:
 8:  // Obtem modulo Mod1
 9:  #define MICO_MODULE_Mod1
10:  struct Mod1 {
11:    #include "module.h"
11:  };
12:  #undef MICO_MODULE_Mod1
13:
14:  // Obtem Mod2
15:  #define MICO_MODULE_Mod2
16:  struct Mod2 {
17:    #include "module.h"
18:  };
19:  #undef MICO_MODULE_Mod2
20:
21:  // Obtem definicoes globais em module.h
22:  #define MICO_MODULE__GLOBAL
23:  #include "module.h"
24:  #undef MICO_MODULE__GLOBAL
25:  #undef MICO_NO_TOPLEVEL_MODULES
```

Neste exemplo assume-se que as definições estão localizadas no arquivo `module.h`. Primeiro você precisa definir `MICO_NO_TOPLEVEL_MODULES` que simplesmente significa que você deseja ler as definições por si mesmo (linha 6). Para cada módulo de alto nível `XYZ` em um arquivo IDL existe um define denominado `MICO_MODULE_XYZ`. A definição deste define ativará todas as definições que pertencem ao módulo `XYZ` (linhas 1 e 15). Não se esqueça de retirar as definições depois de leitura (linhas 12 e 19). Existem algumas definições globais que não pertencem a nenhum módulo. Para estas definições existe um define especial denominado `MICO_MODULE__GLOBAL` (linha 22). A última coisa a ser feita é retirar a definição `MICO_MODULE__GLOBAL` e `MICO_NO_TOPLEVEL_MODULE` (linhas 24 e 25). Este exemplo pode ser encontrado no diretório `mico/test/idl/10`.

Exceções

Devido ao suporte limitado para exceções nas versões mais iniciais do compilador C++ da GNU (a saber gcc 2.7.2) MICO dá suporte a vários tipos de tratamento de exceção:

- tratamento de exceção em conformidade com CORBA
- tratamento de exceção específica de MICO
- sem tratamento de exceção

Dois problemas comuns com tratamento de exceção são ``capturando por classe base" and ``exceções em bibliotecas compartilhadas":

- capturando por classe base: quando uma exceção `X` é lançada deve ser possível capturá-la especificando uma classe base de `X` no catch. Alguns compiladores (ex.: gcc 2.7) não dão suporte a isto.
- exceções em bibliotecas compartilhadas: o lançamento de uma exceção a partir de uma biblioteca compartilhada em uma biblioteca não compartilhada não funciona com alguns compiladores em algumas plataformas (gcc 2.7, gcc 2.8 e egcs 1.x em algumas plataformas).

O tipo de tratamento de exceção usada é determinada pelas capacidades do compilador C++ e opções da linha de comando passada ao script `configure`. Por default *CORBA compliant exception handling* (tratamento de exceção em conformidade com CORBA) será selecionado se o compilador C++ der

suporte à captura por classe base, do contrário *MICO specific exception handling* (tratamento de exceção específica do MICO) será selecionado se o compilador não der suporte a tratamento de exceção. Se as exceções na biblioteca compartilhada não funciona então *no exception handling* (sem tratamento de exceção) é selecionado para o código nas biblioteca compartilhadas. Você pode reforçar *MICO specific exception handling* especificando `-disable-std-eh` como uma opção de linha de comando para `configure`. Você pode desativar o tratamento de exceção especificando `-disable-except` como uma opção de linha de comando para o `configure`.

Você pode descobrir sobre o suporte ao tratamento de exceção dos arquivos executáveis MICO executando o compilador IDL com a opção de linha comando `-config`:

```
$ idl --config
MICO version: 2.2.7
supported CORBA version: 2.2
exceptions: CORBA compliant
modules are mapped to: namespaces
STL is: miniSTL
SSL support: no
loadable modules: yes
```

Como o nome já indica este modo de tratamento de exceção está em conformidade com a especificação CORBA. Você pode usar `throw` para lançar exceções. As exceções são capturadas especificando o tipo exato ou uma dos tipos base da exceção. Eis alguns exemplos:

```
// throw CORBA::UNKNOWN exception
throw CORBA::UNKNOWN();
// catch CORBA::UNKNOWN exception
try {
  ...
} catch (CORBA::UNKNOWN &ex) {
  ...
}
// catch all system exceptions (including CORBA::UNKNOWN)
try {
  ...
} catch (CORBA::SystemException &ex) {
  ...
}
// catch all user exceptions (wont catch CORBA::UNKNOWN)
try {
  ...
} catch (CORBA::UserException &ex) {
  ...
}
// catch all exceptions (including CORBA::UNKNOWN)
try {
  ...
} catch (CORBA::Exception &ex) {
  ...
}
```

Se uma exceção é lançada mas não é capturada MICO imprimirá uma descrição curta da exceção e terminará o processo. Alguns compiladores C++ não dão suporte apropriado às exceções nas bibliotecas compartilhadas, outras não dão suporte algum a exceções. Neste caso o tratamento de exceção não está disponível nas bibliotecas compartilhadas ou não estão disponíveis, respectivamente. As palavras reservadas relacionadas a tratamento de exceção (try, throw, catch) não pode ser usado

neste modo. Pode-se usar mico_throw() mas somente imprimirá uma descrição breve da exceção passada e terminará o processo.