XcodeML/C++ 仕様書

V1.0J (October 15, 2015)

XcalableMP/Omni Compiler Project

改版履歴

XcodeML/C Version 0.91J

- 配列要素の参照の XML 要素を変更。
- subArrayRef 要素を変更。
- indexRange 要素を追加。

XcodeML/C++ draft 0.1J

- C++対応ドラフト初版

XcodeML/C++ 1.0J

- C++対応初版。2015/10/15

目次

1　はじめに

この仕様書は、C 言語および C++言語の中間表現である XcodeML を記述する。XcodeML は、以下の特徴を持つ。

- 入力された C 言語または C++言語のプログラムを意味的（semantic）に復元可能な情報を保持する。
- C++言語相当の型情報を、シンボル情報から独立した形で表現する。
- human-readable なフォーマット(XML)をもつ。

2　翻訳単位と XcodeProgram 要素

ソースファイルに、＃include 指定されたファイルを再帰的にすべて展開したものを、翻訳単位と呼ぶ。翻訳単位は XcodeProgram 要素で表現される。

```
<XcodeProgram>
  typeTable 要素（3 章）
  globalSymbols 要素（4.2 節）
  globalDeclarations 要素（5.1 節）
</XcodeProgram>
```

属性（optional）:　compiler-info, version, time, language, source

XcodeML ファイルのトップレベルの XML 要素は、XcodeProgram 要素である。XcodeProgram 要素は以下の子要素を含む。

- typeTable 要素 – プログラムで利用されているデータ型の情報
- globalSybmols 要素 – プログラムで利用されている大域変数の情報
- globalDeclarations 要素 – 関数、変数宣言などの情報

XcodeProgram 要素は、属性として以下の情報を持つことができる

- compiler-info － CtoC コンパイラの情報
- version － CtoC コンパイラのバージョン情報
- time － コンパイルされた日時
- language － ソース言語情報
- source － ソース情報

2.1　name 要素

変数名や、タイプ名などの名前を指定する XML 要素である。

```
<name>[::][スコープ名::]…名前</name>
```

属性（optional）: type

内容は、スコープ修飾子とスコープ解決演算子を含む名前を表現し、0個以上のスコープ名と「::」の組に続く名前を文字列として持つ。文字列は空白文字を含んではならない。名前は変数名だけでなく、スコープ名、typedef 名、using 文による型の別名（C++）にもなれる。

type 属性の値はデータ型識別名である。

備考：

以下のような属性を考えたが、検討保留する。トランスレータでどのような名前のユニーク化が必要になるか今の段階では分からないため。また、using 文に関わる名前の正規化は、using 文のネストが同じ名前空間を別経路で複数回取り込む場合があるため、難しいかもしれない。

- fullName 属性は、0 個以上のスコープ名と「::」の組に続く名前を文字列として持つ。文字列は空白文

字を含んではならない。スコープ名と「::」は、そのオブジェクトのスコープ名を省略せずすべて書き下した形でなければならない。また、スコープ名に namespace の別名定義があるとき、元のスコープ名に正規化した形で表現される。using 文があるとき、スコープ名は省略しない形に正規化される。

例:

以下のプログラムで、

```
namespace NS {
  int a;                    // (1)
}
NS::a = 10;                 // (2)
```

(1)における a は、以下のように表現される。

```
<name type="int">a</name>
```

(2)における a は、以下のように表現される。

```
<name type="int" fullName="NS::a">NS::a</name>
```

例:

以下のプログラムで、

```
struct S {
  int data;
  int foo(int n) { return n + 1; }
};

int S :: *d = &S :: data;               // (1)
int (S :: *f)(int) = &S :: foo;         // (2)
struct S s1;
int *p = &s1.data;                      // (3)
```

(1),(2)の d と f の名前は、それぞれ以下のように表現される。P1は int へのポインタ型である。

```
<name type="P1">S::d</name>
<name type="P1">S::f</name>
```

(1)と(2)の右辺式は、それぞれ以下のように表現される。S は変数でないので memberAddr 要素は用いられず、data 変数のスコープと解釈する。

```
<varAddr type="P0" scope="global">S::data</varAddr>
<varAddr type="P0" scope="global">S::foo</varAddr>
```

(3)の右辺式は、以下のように表現される。s1 は変数名なので、s1.data は memberAddr 要素で表現される。

```
<memberAddr type="P5" member="data">
  <varAddr type="P4" scope="global">s1</varAddr>
</memberAddr>
```

2.2 value 要素

初期値の式を表現する。

```
<value>
  [ 式の要素(7 章) or value 要素
  … ]
</value>
```

属性: なし

{} で囲まれた式の並びは、value 要素のネストで表現する。

例:

int 型の初期値 1 に対応する表現は次のとおりになる。

```
<value>
  <intConstant type="int">1</intConstant>
</value>
```

int 型配列の初期値 { 1, 2 } に対応する表現は次のとおりになる。

```
<value>
  <value>
    <intConstant type="int">1</intConstant>
    <intConstant type="int">2</intConstant>
  </value>
</value>
```

3　typeTable 要素とデータ型定義要素

typeTable 要素は、翻訳単位(2 章)に対して一つだけ存在し、翻訳単位で使われているすべてのデータ型についての情報を定義する。

```
<typeTable>
  [ データ型定義要素
  … ]
</typeTable>
```

属性(optional): なし

typeTable 要素は、翻訳単位を表現する XcodeProgram 要素(2 章)の直接の子要素であり、データ型を定義するデータ型定義要素の列からなる。データ型定義要素には以下の要素がある。

- basicType 要素(3.4 節)
- pointerType 要素(3.5 節)
- functionType 要素(3.6 節)
- arrayType 要素(3.7 節)
- structType 要素と unionType 要素(3.8 節)(廃止予定)
- class 要素(3.9 節)
- enumType 要素(3.10 節)
- typeInstance 要素(9.2 節)
- classTemplate 要素(8.2 節)
- aliasTemplate 要素(8.4 節)

すべてのデータ型定義要素は、型識別名(3.1 節)を表す type 要素をもつ。
データ型定義要素は、データ型定義要素属性(0 節)をもつことができる。

要検討：dec|type 対応

dec|type(式) は式の型を表すが、式はスコープをもつので typeTable の中に移動することができない。

- 案 1: 式の中のすべての名前に、スコープ名を付ける。decltype(main:x + main:y) など。→とても煩雑。scopename を持たない {} の中に出現した場合は？
- 案 2: typeTable を翻訳単位に一つにするのではなく、スコープ毎にもつようにする。
- 案 3: decltype が出現したスコープに限り、typeTable をもつ。

3.1　データ型識別名

プログラム内において、データ型はデータ型識別名で区別される。その名前は、次のいずれかである。

- 基本データ型(3.4 節)
  - C, C++言語の基本データ型(C++拡張)

    'void', 'char', 'short', 'int' , 'long', 'long_long', 'unsigned_char', 'unsigned_short', 'unsigned', 'unsigned_long', 'unsigned_long_long', 'float', 'double', 'long_double', 'wchar_t', 'char16_t',

'char32_t', 'bool' (_Bool 型)

- _Complex、_Imaginary に対応する型

  'float_complex', 'double_complex', 'long_double_complex', 'float_imaginary', 'double_imaginary', 'long_double_imaginary'

- GCC の組み込み型

  '__builtin_va_arg'

- 型推論の対象(C++)

  'auto'

- 名前空間の名前(C++)

  'namespace'

- 型の抽象(C++) － テンプレートの型仮引数の型の名前

  'any_class', 'any_typename'

- 派生データ型とクラス

  他のデータ型識別名とは異なる、翻訳単位内でユニークな英数字の並び。

3.2 typeName 要素

typeName 要素はデータ型識別名を表す。

```
<typeName/>
```

属性(必須): ref

属性(optional): access

以下の属性をもつことができる。

- ref 属性 － データ型識別名を示す。
- access 属性 － inheritedFrom 要素の子要素のときだけ使用する。public, private または protecded のいずれかの値をとる。

typeName 要素は以下のように使用される。

- 型を引数とする関数の呼出しで
  - sizeOfExpr(7.11 節)、gccAlignOfExpr(7.11 節)、builtin_op(11.5 節)
- テンプレートの定義の型仮引数として(8 章)
- テンプレートのインスタンスの型実引数として(9 章)
- 構造体とクラスの継承元(3.9.1 項)

例: 式 sizeof(int) は以下のように表現される。

```
<sizeOfExpr>
  <typeName ref="int"/>
</sizeOfExpr>
```

3.3 データ型定義要素属性

データ型定義要素は共通に以下の属性を持つことができる。これらをデータ型定義要素属性と呼ぶ。

- is_const － そのデータ型が const 修飾子をもつかどうか
- is_volatile － そのデータ型が volatile 修飾子をもつかどうか
- is_restrict － そのデータ型が restrict 修飾子をもつかどうか
- is_static － そのデータ型が static 属性をもつかどうか
- reference(C++) － 属性値が lvalue のとき左辺値参照、rvalue のとき右辺値参照を意味する。属性値が default または属性が省略されているとき文脈依存であることを意味するが、lvalue または rvalue の値をもつことが望ましい。引数の値渡し（通常の場合）に対しては、defalut とする。
- access（C++） － アクセス指定子に対応。"private", "protected"または"public"
- is_virtual（C++） － そのメンバ関数が virtual 属性をもつかどうか。
- is_userDefined（C++） － その演算がユーザ定義によりオーバーロードされているかどうか

”is_” で始まる属性の値には、真を意味する 1 と true、および、偽を意味する 0 と false が許される。属性が省略されたとき、偽を意味する。

例: 左辺値参照と右辺値参照

以下の参照（左辺値参照）の宣言があるとき、

```
int& n_alias = n_org;
```

変数 n_alias のデータ型識別要素は以下のようになる。

```
<basicType type="B0" name="int" reference="lvalue"/>
```

以下のコンストラクタ（ムーブコンストラクタ）の定義の引数に現れた右辺値参照について、

```
struct Array {
  int *p, len;
  Array( Array&& obj ) : p(obj.p), len(obj.len) {
    obj.p = nullptr;   obj.len = 0;
  }
}
```

仮引数 obj のデータ型識別要素は以下のようになる。

```
<basicType="B1" name="B2" reference="rvalue"/>
```

3.4 basicType 要素

basicType 要素は、他のデータ型識別要素にデータ型定義要素属性を加えた、新しいデータ型定義要素を定義する。

```
<basicType/>
```

属性（必須）: type, name

属性(optional): alignas, **データ型定義要素属性**

以下の属性を持つ。

- type ― この型に与えられたデータ型識別名
- name ― この型の元になる型のデータ型識別名

備考: 旧仕様と実装の違い

本仕様は、旧仕様とは異なり、実装に合わせた。旧仕様では以下のように定義されていた。

basicType 要素は、C,C99 の基本データ型を定義する。

実装では、データ型定義要素属性（const など）を持たない基本データ型に対応するデータ型識別要素は定義されず、type="int" のようにデータ型識別名だけで表現されている。また、基本データ型以外の型（構造型など）に属性を付ける場合に、basicType 要素を使用している。

例:

```
struct {int x; int y;} s;
struct s const * volatile p;
```

は次の XcodeML に変換される。 basicType 要素によって、"struct s const"を意味するデータ型識別名 B0 が定義されている。

```
<structType type="S0">
  <symbols>
    <id type="int"><name="x"><./name></id>
    <id type="int"><name="y"><./name></id>
  </symbols>
</structType>
<basicType type="B0" is_const="1" name="S0"/>
<pointerType type="P0" is_volatile="1" ref="B0"/>
```

3.5 pointerType 要素

pointerType 要素はポインタのデータ型を定義する。

```
<pointerType/>
```

属性(必須): type, ref

属性(optional): データ型定義要素属性

以下の属性を持つ。

- type ― この型に与えられたデータ型識別名
- ref ― このポインタが指すデータのデータ型識別名

pointerType 要素は、子要素を持たない。

例:

"int *" に対応するデータ型定義は以下のようになる。

```
<pointerType type="P0123" ref="int"/>
```

3.6 functionType 要素

funtionType 要素は、関数型を定義する。

```
<functionType>
  [ params 要素(5.3.4 節) ]
</functionType>
```

属性(必須): type, return_type

属性(optional): is_inline

- type ― この関数型に与えられたデータ型識別名
- return_type ― この関数型が返すデータのデータ型識別名
- is_inline ― この関数型が inline 型であるかどうかの情報、0 または 1、false または true
  省略時は 0, false を意味する。

プロトタイプ宣言がある場合には、引数の XML 要素に対応する params 要素を含む。

例:

"double foo(int a, int b)" の foo に対するデータ型は以下のようになる。

```
<functionType type="F0457" return_type="double">
  <params>
    <name type="int">a</name>
    <name type="int">b</name>
  </params>
</fucntionType>
```

3.7 arrayType 要素

arrayType 要素は、配列データ型を定義する。

```
<arrayType>
  [ arraySize 要素
  … ]
</arrayType>
```

属性(必須): type, element_type

属性(optional): array_size, データ型定義要素属性

arrayType 要素は以下の属性を持つ。

- type ー この配列型に与えられたデータ型識別名
- element_type ー 配列要素のデータ型識別名
- array_size ー 配列のサイズ(要素数)。array_size と子要素の arraySize を省略した場合は、サイズ未指定を意味する。array_size 属性は子要素の arraySize と同時に指定することはできない。

以下の子要素を持つ。

- arraySize ー 配列のサイズ(要素数)を表す式。式要素ひとつを子要素に持つ。
  サイズを数値で表現できない場合や、可変長配列の場合に指定する。arrayType 要素が arraySize 要素を持つ場合、array_size 属性の値は"*"とする。

例:

"int a[10]"の a に対する type_entry は以下のようになる。

```
<arrayType type="A011" element_type="int" array_size="10"/>
```

3.8 unionType 要素

union(共用体)データ型は、unionType 要素で定義する。

```
<unionType>
  symbols 要素
</unionType>
```

属性(必須): type

属性(optional): データ型定義要素属性

unionType 要素は以下の属性を持つ。

- type ー この共用体型のデータ型識別名

unionType 要素は、メンバに対する識別子の情報である symbols 要素を持つ。 構造体・共用体のタグ名がある場合には、スコープに対応するシンボルテーブルに定義されている。

メンバのビットフィールドは、id 要素の bit_field 属性または id 要素の子要素 である bitField 要素に記述する(4.1 節)。

3.9 structType 要素と class 要素 (C++)

構造体およびクラスを表現する。

```
<structType> or <class>
  [ inheritedFrom 要素(3.9.1) ]
```

```
    symbols 要素(4.3 節)
    [ declarations 要素(5.2 節) ]
</structType> or </class>
```

属性(必須): type

属性(optional): lineno, file, inherited, **データ型定義要素属性**

以下の子要素をもつ。

- symbols 要素 － メンバ変数名とメンバ関数名のリスト
- declarations 要素 － メンバ変数とメンバ関数の宣言、メンバ関数の定義、ディレクティブ文字列を含む

以下の属性をもつ。

- type(必須) － この構造体、共用体またはクラスに与えられたデータ型識別名
- inherited － 継承元のクラスの名前。name 要素の名前と同様に、スコープ名付きの名前にもなれる。

メンバのビットフィールドは、id 要素の bit_field 属性または id 要素の子要素 である bitField 要素に記述する(4.1 節)。

構造体またはメンバの名前は、同じ type 属性をもつ id 要素で指定する。typedef 文または using 文で指定された別名もまた、同じ type 属性をもつ id 要素で指定する。

要検討:

friend 関数の宣言。friend 関数はそのクラスのメンバ関数ではない。

例:

以下の構造体宣言

```
struct {
  int x;
  int y : 8;
  int z : sizeof(int);
};
```

に対する structType 要素は以下のようになる。この構造体のデータ型識別名は S0 と定義された。

```
  <stuctType type="S0">
    <symbols>
      <id type="int">
        <name>x</name>
      </id>
      <id type="int" bit_field="8">
        <name>y</name>
      </id>
```

```
        <id type="int" bit_field="*">
          <name>z</name>
          <bitField>
            <sizeOfExpr>
              <typeName ref="int"/>
            </sizeOfExpr>
          </bitField>
        </id>
      </symbols>
    </structType>
```

3.9.1 inheritedFrom 要素 (C++)

structType 要素と class 要素だけの子要素であり、継承元の構造体またはクラスの並びを表現する。

```
<inheritedFrom>
  [ typeName 要素(3.2 節)
  ... ]
</inheritedFrom>
```

属性なし

以下の子要素をもつ。

- typeName 要素 － 継承する構造体またはクラスのデータ型識別名を示す。access 属性により、public, private または protected の区別を指定できる。

3.10 enumType 要素

enum 型は、enumType 要素で定義する。type 要素で、メンバの識別子を指定する。

```
<enumType>
  [ name 要素 ]
  symbols 要素
</enumType>
```

属性(必須): type

属性(optional): データ型定義要素属性

次の子要素を持つ。

- symbols 要素 － メンバの識別子を定義する。メンバの値は id 子要素の value 子要素で表す。
- name 要素(C++、オプショナル) － スコープ付き列挙型のときのスコープ名を定義する。

メンバの識別子は、スコープに対応するシンボルテーブルにクラス moe として定義されている。 enum のタグ名がある場合には、スコープに対応するシンボルテーブルに定義されている。

例:

"enum { e1, e2, e3 = 10 } ee; "の ee に対する enumType 要素は以下のようになる。

```
<enumType name="E0">
  <symbols>
    <id>
      <name>e1</name>
    </id>
    <id>
      <name>e2</name>
    </id>
    <id>
      <name>e3</name>
      <value><intConstant>10</intConstant></value>
    </id>
  </symbols>
</enumType>
```

3.11 parameterPack 要素 （C++）

可変長引数を表現するための、仮引数の並びに対応する。

```
<parameterPack/>
```

属性(必須): type, element_type

属性(optional): **データ型定義要素属性**

以下の属性を持つ。

- type ― パックされた型に与えられたデータ型識別名
- elem_type ― パックされる個々の型のデータ型識別名

parameterPack 要素は、子要素を持たない。

例:

以下の関数テンプレートの定義において、

```
template<typename T1, typename ... Types>
T1 product(T1 val1, Types ... tail) {
  return val1 * product(tail...);
}
```

"typename ... Types" に対応するデータ型定義は以下のようになる。

```
<parameterPack type="K0" ref="typename"/>
```

4　シンボルリスト

4.1　id 要素

id 要素は、変数名や配列名、関数名、struct/union のメンバ名、 関数の引数、compound statement の局所変数名を定義する。

```
<id>
  name 要素（2.1 節）
  [ value 要素（2.2 節） ]
  [ bitField 要素 ]
  [ alignAs 要素 ]
</id>
```

属性（optional）: sclass, fspec, type, bit_field, align_as, is_gccThread, is_gccExtension

id 要素は次の属性を持つことができる。

- sclass 属性　－　storage class をあらわし、 'auto', 'param', 'extern', 'extern_def', 'static', 'register', 'label', 'tagname', 'moe', 'typedef_name', 'template_param'（C++、テンプレートの型仮引数名）, 'namespace_name'（C++）, 'thread_local'（C++）, 'alias_name'（C++、using 文による別名）のいずれか。
- fspec 属性　－　function specifier をあらわし、'inline', 'vertual', 'explicit'のいずれか。
- 【要検討】storage class specifier 以外の decl-specifier である 'friend', 'constexpr'もここで表現するか？
- type 属性　－　識別子のデータ型()
- bit_field 属性　－　structType、unionType と class 要素においてメンバのビットフィールドを数値で指定する。
- align_as 属性　－　structType、unionType と class 要素において、メンバの alignment を数値またはデータ型識別名で指定する。
- is_gccThread 属性　－　GCC の__thread キーワードが指定されているかどうかの情報、0 または 1、false または true。
- is_gccExtension 属性

以下の子要素を持つことができる。

- name 要素　－　識別子の名前は name 要素で指定する。
- value 要素　－　識別子に対応した値は value 要素で指定する。const 変数の初期化（const int a=式）、参照の初期化（int& a=初期化子）、namespace の別名定義（namespace new = scope1::old）などをここで表現する。
  要検討: 実装時に再検討。何もかも value 要素にするのがよいか？
- bitField 要素　－　unionType と class 要素においてメンバのビットフィールドの値を bit_field 属性の数値として指定できないとき使用する。bitField 要素は式を子要素に持つ。bitField 要素を使用するとき、

bit_field 属性の値は、"*" とする。

- alignAs 要素 － structType、unionTypeとclass 要素においてメンバの alignmentを align_as 属性の数値として指定できないとき、alignAs 要素の子要素として式の要素で指定する。

識別子が変数などの場合、そのアドレスをvalue要素として持つ。コンパイラで生成される変数などの場合は、なくてもよい。

要確認：

この1行は実態に合っているのか？ value 要素として持つのは自然か？

例:

"int xyz;"の変数 xyz に対するシンボルテーブルエントリは以下のようになる。なお、P6e7e0 は "int *"に対する type_id。

```
<id sclass="extern_def" type="int">
 <name>xyz</name>
 <value>
   <VarAddr type="P6e7e0">xyz</varAddr>
 </value>
</id>
```

"int foo()"の関数 foo に対するシンボルテーブルエントリは以下のようになる。なお、F6f168 は、foo のデータ型に対する type_id。P6f1a8 は、F6f168 へのポインタの type_id。識別子 foo は関数へのポインタになることに注意。

```
<id sclass="extern_def" type="0x6f168">
 <name>foo</name>
 <value>
  <funcAddr type="0xfla8">foo</funcAddr>
 </value>
</id>
```

4.2 globalSymbols 要素

大域のスコープを持つ識別子を定義する。

```
<globalSymbols>
  [ id 要素(4.1 節)
    … ]
</globalSymbols>
```

属性なし

子要素として、大域のスコープを持つ識別子の id 要素の並びを持つ。

4.3 symbols 要素

局所スコープを持つ識別子を定義する。

```
<symbols>
  [ id 要素(4.1 節)
    … ]
</symbols>
```

属性なし

子要素として、定義する識別子に対する id 要素を持つ。

5　globalDeclarations 要素と declarations 要素

5.1　globalDeclarations 要素

大域的な(翻訳単位全体をスコープとする)変数、関数などの宣言と定義を行う。

```
<globalDeclarations>
  [{  varDecl 要素(5.4 節)  or
      functionDecl 要素(5.5 節)  or
      usingDecl 要素(5.6 節)  or
      functionDefinition 要素(5.3 節)  or
      functionTemplate 要素(8.3 節)  or
      text 要素(6.21 節) }
  … ]
</globalDeclarations>
```

属性なし

以下の子要素を持つ。

- functionDefinition 要素　－　関数の定義
- varDecl 要素　－　変数の定義
- functionDecl 要素　－　関数の宣言
- text 要素　－　ディレクティブなど任意のテキストを表す

5.2　declarations 要素

compoundStatement(6.2 節)、class(3.9 節)などをスコープとする変数、関数などの宣言と定義を行う。

```
<declarations>
  [{  varDecl 要素(5.4 節)  or
      functionDecl 要素(5.5 節)  or
      usingDecl 要素(5.6 節)  or
      functionDefinition 要素(5.3 節)  or
      text 要素(6.21 節) }
  … ]
</declarations>
```

属性なし

以下の子要素を持つ。

- functionDefinition 要素　－　関数の定義
- varDecl 要素　－　変数の定義
- functionDecl 要素　－　関数の宣言

- text 要素 － ディレクティブなど任意のテキストを表す

5.3 functionDefinition 要素

関数定義、メンバ関数の定義、コンストラクターの定義、デストラクターの定義、および、演算子オーバーロードの定義を行う。以下のいずれか一つの子要素を持つ。

```
<functionDefinition>
  name 要素(2.1 節) or operator 要素(5.3.1) or constructor 要素(5.3.2) or descructor 要素(5.3.3)
  symbols 要素(4.3 節)
  params 要素(5.3.4)
  body 要素
</functionDefinition>
```

属性(optional): is_gccExtension

以下のいずれか一つの子要素を持つ。

- name 要素 － 関数またはメンバ関数のときの、関数の名前
- operator 要素 － 演算子オーバーロードのときの、演算子の名前
- constructor 要素 － 構造体またはクラスのコンストラクタのとき
- destructor 要素 － 構造体またはクラスのデストラクタのとき

加えて、以下の子要素をもつ。

- symbols 要素 － パラメータ(仮引数)のシンボルリスト。子要素は id 要素の並び。
- params 要素 － パラメータ(仮引数)の並び
- body 要素 － 関数本体。子要素として文(通常は compoundStatement)を含む。関数に局所的な変数などの宣言は、body 要素の中に記述される。body 要素内に GCC のネストされた関数を表す functionDefinition を含む場合がある。

以下の属性を持つ

- is_gccExtension 属性

例：

関数の定義

```
struct sss *foo(struct sss *arg1, int nnn)
{
  ・・・(略)・・・
}
```

に対し、以下の表現が対応する。

```
<functionDefinition>
  <name>foo</name>
```

```
    <symbols>
      <id type="P1" sclass="param">
        <name>arg1</name>
      </id>
      <id type="int" sclass="param">
        <name>nnn</name>
      </id>
    </symbols>
    <params>
      <name type="P1">arg1</name>
      <name type="int">nnn</name>
    </params>
    <body>
      <compoundStatement>
        …(略)…
      </compoundStatement>
    </body>
  </functionDefinition>
```

5.3.1 operator 要素 (C++)

functionDefinition 要素の子要素。演算子オーバーロードを定義するとき、name 要素の代わりに指定する。

```
<operator>演算子名</operator>
```

属性なし

演算子名には、単項演算要素名(7.11 節)、二項演算要素名(7.10 節)などの 7 章で定義される演算子の XML 要素の名前、または、ユーザ定義リテラルのアンダースコアで始まる名前を記述する。以下に例示する。

```
<operator>plusExpr</operator>
<operator>_my_op</operator>
```

5.3.2 constructor 要素 (C++)

functionDefinition 要素の子要素。そのメンバ関数がコンストラクタのとき、name 要素の代わりに指定する。

```
<constructor>
  [{ name 要素(2.1 節)
     value 要素(2.2 節) }
  … ]
</constructor>
```

属性(optional): is_explicit

name 要素と value 要素の組は初期化構文に対応する。

要検討： コンストラクタのバリエーションに対応し切れていない。

5.3.3 destructor 要素 (C++)

functionDefinition 要素の子要素。そのメンバ関数がデストラクタであるとき、name 要素の代わりに指定する。

```
<destructor/>
```

5.3.4 params 要素

関数の引数の並びを指定する。

```
<params>
  [ { name 要素(2.1 節)
    [ value 要素(2.2 節) ] }
  … ]
  [ ellipsis ]
</params>
```

属性なし

以下の子要素をもつことができる。

- name 要素 － 引数の名前に対応する name 要素を持つ。引数のデータ型の情報は、name 要素の type 属性名と同じ type 属性名をもつデータ型定義要素(3 章)で表現される。
- value 要素 － params が関数またはラムダ関数の仮引数並びで、直前の name 要素に対応する仮引数がデフォルト実引数をもつとき、それを表現する。
- ellipsis － 可変長引数を表す。params の最後の子要素に指定可能。

params 要素内の name 要素は、引数の順序で並んでいなくてはならない。

5.4 varDecl 要素

変数の宣言を行う。

```
<varDecl>
  name 要素(2.1 節)
  [ value 要素(2.2 節) ]
</varDecl>
```

属性なし

変数宣言を行う識別子の名前を name 要素で指定する。 以下の子要素を持つ。

- name 要素 － 宣言する変数に対する name 要素を持つ。
- value 要素 － 初期値を持つ場合、value 要素で指定する。配列・構造体の初期値の場合、value 要素に複数の式を指定する。

例:

```
int a[] = { 1, 2 };
<varDecl>
  <name>a</name>
  <value>
    <intConstant type="int">1</intConstant>
    <intConstant type="int">2</intConstant>
  </value>
</varDecl>
```

5.5 functionDecl 要素

関数宣言を行う。

```
<functionDecl>
  name 要素(2.1 節)
</functionDecl>
```

属性なし

以下の子要素を持つ

- name 要素 － 関数名を指定する

5.6 usingDecl 要素 (C++)

C++の using 宣言(using declaration)と using 指示(using directive)に対応する。

```
<usingDecl>
  name 要素(2.1 節)
</usingDecl>
```

属性(optional): lineno, file, namespace

以下のように using 文に対応する。

- using 指示 ”using namespace 名前空間名” の形のとき
  - namespace 属性の値を 1 または true とする。
  - 名前空間名を name 要素とする。名前空間名にはスコープ名と「::」が含まれることがある。

- using 宣言 "using 名前” の形のとき
  - namespace 属性を持たないか、値を 0 または false とする。
  - 名前を name 要素とする。名前にはスコープ名と「::」が含まれることがある。
- 別名宣言 "using 別名 = 型” の形のとき、usingDecl 要素では表現されない。typedef と同様、データ型定義要素（3 章）で表現される。

6　文の要素

C の文の構文要素に対応する XML 要素である。それぞれの XML 要素には、文の元の行番号とファイル名を属性として付加することができる。

- lineno　－　文番号を値として持つ
- file　－　この文が含まれているファイル名

6.1　exprStatement 要素

式で表現される文を表す。式の要素（7 章）を持つ。

```
<exprStatement>
  式の要素（0 章）
</exprStatement>
```

属性(optional): lineno, file

6.2　compoundStatement 要素

複文を表現する。

```
<compoundStatement>
  symbols 要素（4.3 節）
  declarations 要素（5.2 節）
  body 要素
</compoundStatement>
```

属性(optional): lineno, file

以下の子要素を持つ。

- symbols 要素　－　このスコープの中で定義されているシンボルリスト
- declarations 要素　－　このスコープの中で定義される宣言
- body 要素　－　複文本体。文の要素の並び。

6.3　ifStatement 要素

if 文を表現する。

```
<ifStatement>
  condition 要素
  then 要素
  else 要素
</ifStatement>
```

属性(optional): lineno, file

以下の子要素を持つ。

- condition 要素 － 条件式を子要素として含む
- then 要素 － then 部の文を子要素として含む
- else 要素 － else 部の文を子要素として含む

6.4 whileStatment 要素

while 文を表現する。

```
<whileStatement>
  condition 要素
  body 要素
</whileStatement>
```

属性(optional): lineno, file

以下の子要素を持つ

- condition 要素 － 条件式を子要素として含む
- body 要素 － 本体の文を子要素として含む

6.5 doStatement 要素

do 文を表現する。

```
<doStatement>
  body 要素
  condition 要素
</doStatement>
```

属性(optional): lineno, file

以下の子要素を持つ。

- body 要素 － 本体を表す、文の要素の並びを含む
- condition 要素 － 条件式を表す式の要素を含む

6.6 forStatement 要素

for 文(従来仕様)を表現する。

```
<forStatement>
  [ init 要素 ]
  [ condition 要素 ]
  [ iter 要素 ]
```

```
  body 要素
</forStatement>
```

属性(optional): lineno, file

以下の要素を持つ。

- init 要素 － 初期化式または宣言文を要素として含む
- condition 要素 － 条件式として式の要素を含む
- iter 要素 － 繰り返し式として式の要素を含む
- body 要素 － for 文の本体を表す、文の要素の並びを含む。

init 要素は、for 文の中の初期化式または宣言文を表現する。

```
<init>
  式の要素 or symbols 要素
</init>
```

属性なし

init 要素は、forStatement 要素の中だけに現れる。初期化式を意味する式の要素を含むか、または、0 個以上の局所変数の宣言を意味する symbols 要素を含む。

6.7 rangeForStatement 要素 (C++)

C++仕様の for 文

for (*for-range-declaration* : *expression*) *statement*

を表現する。

```
<rangeForStatement>
  id 要素
  range 要素
  body 要素
</rangeForStatement>
```

属性(optional): lineno, file

以下の子要素を持つ。

- id 要素(4.1 節)
- range 要素 － 配列やコンテナを表す式の要素(7 章)を含む。
- body 要素 － for 文の本体を表す、文の要素(6 章)の並びを含む。

6.8 breakStatement 要素

break 文を表現する。

```
<breakStatement/>
```

属性(optional): lineno, file

6.9 continueStatement 要素

continue 文を表現する。

```
<continueStatement/>
```

属性(optional): lineno, file

6.10 returnStatment 要素

return 文を表現する。

```
<returnStatement>
  [ 式の要素 ]
</returnStatement>
```

属性(optional): lineno, file

return する式を、子要素として持つことができる。

6.11 gotoStatement 要素

goto 文を表現する。

```
<gotoStatement>
  name 要素 or 式の要素
</gotoStatement>
```

属性(optional): lineno, file

子要素に name 要素か式のいずれかを持つ。式は GCC においてジャンプ先として指定可能なアドレスの式を表す。

- name 要素 － ラベル名の名前を指定する。
- 式の要素 － ジャンプ先のアドレス値を指定する。

6.12 tryStatement 要素（C++）

try 構文を表現する。

```
<tryStatement>
  body 要素
</tryStatement>
```

属性(optional): lineno, file

以下の子要素を持つ。

- body 要素 － 本体を表す、文の要素(6 章)の並びを含む

6.13 throwStatement 要素 (C++)

throw 文を表現する。

```
<throwStatement/>
```

属性(optional): lineno, file

子要素を持たない。

6.14 catchStatement 要素 (C++)

catch 構文を表現する。

```
<catchStatement>
  params 要素(5.3.4 節)
  body 要素
</catchStatement>
```

属性(optional): lineno, file

以下の子要素を持つ。

- params 要素 － 内容は 1 つの name 要素または 1 つの ellipsis でなければならない。補足する例外の型を示す。
- body 要素 － 本体を表す、文の要素(6 章)の並びを含む

6.15 statementLabel 要素

goto 文のターゲットのラベルを表す。

```
<statementLabel>
  name 要素
</statementLabel>
```

属性なし

ラベル名を name 要素として持つ。

- name 要素 － ラベル名の名前を指定する。

6.16 switchStatement 要素

switch 文を表現する。

```
<statementLabel>
  value 要素
```

```
    body 要素
  </statementLabel>
```

属性(optional): lineno, file

以下の子要素を持つ。

- value 要素 － switch する値を表す式の要素(0 章)
- body 要素 － switch 文の本体を表す文の要素(0 章)であり、多くの場合 compoundStatement 要素(6.2 節)となる。caseLabel 要素(6.17 節)と gccRangedCaseLabel 要素(6.18 節)と defaultLabel 要素(6.19 節)を含むことができる。

6.17 caseLabel 要素

switch 文の case 文を表す。switch 要素の中の body 要素の中の compoundStatement の中だけに現れることができる。

```
<caseLabel>
  value 要素
</caseLabel>
```

属性(optional): lineno, file

case の値を子要素としてもつ。

- value 要素 － case の値を指定する。

6.18 gccRangedCaseLabel 要素

gcc 拡張の case 文での範囲指定を表す。switch 要素の中の body 要素の中の compoundStatement の中だけに現れることができる。

```
<gccRangedCaseLabel>
  value 要素
  value 要素
</gccRangedCaseLabel>
```

属性(optional): lineno, file

case の値を要素としてもつ。

- value 要素 － case の値の下限値を指定する。
- value 要素 － case の値の上限値を指定する。

6.19 defaultLabel 要素

switch 文の default ラベルを表す。switch 要素の中の body 要素の中の compoundStatement の中だけに現れることができる。

```
<defaultLabel/>
```

属性(optional): lineno, file

6.20 pragma 要素

pragma 要素は#pragma 文を表す。

```
<pragma>文字列</pragma>
```

属性(optional): lineno, file

#pragma に指定する文字列を持つ。

6.21 text 要素

text 要素は任意のテキストを表し、コンパイラに依存したディレクティブなどの情報を要素として持つために使用する。

```
<text>文字列</text>
```

属性(optional): lineno, file

内容に任意の文字列を持つ。この要素は globalDeclarasions にも出現する。

7　式の要素

式の構文要素に対応する XML 要素である。式の要素には、本章に記述された XML 要素以外に、以下のものがある。

- functionInstance 要素（9.3 節）

式の要素には、共通して以下の属性を付加できる。

- type 属性 ― 式のデータ型情報を取り出すことができる。
- （廃止予定）lvalue 属性 ― 式が左辺値であることを示す。

要検討：

lvalue 属性は、式の要素の属性からテータ型定義要素の属性に移動したい。

7.1　定数の要素

定数は以下の XML 要素によって表現する。

```
<intConstant>10 進数または 16 進数</intConstant>
<longlongConstant>16 進数 16 進数</longlongConstant>
<floatConstant>浮動小数点数</floatConstant>
<stringConstant>文字列</stringConstant>
<moeConstant>列挙型メンバ名</moeConstant>
<booleanConstant>真偽値</booleanConatant>
<funcAddr>関数名</funcAddr>
```

属性(必須): type

- intConstant 要素 ― 整数の値を持つ定数を表す。数値として、十進数もしくは、16 進数（0x から始まる）を記述する。type 属性には”int”, ”long”, ”unsigned”, ”unsigned_long”, ”char”と”wchar_t”が許される。C++ではこれらに加えて、”char16_t”と”char32_t”が許される。
  備考：char16_t は必ず 16 ビット、char32_t は必ず 32 ビットだが、wchar_t は環境によって 16 ビットまたは 32 ビットであると定義されている。
- longlongConstant 要素 ― 32 ビット 16 進数(0x から始まる)の2つの数字を空白で区切って記述する。type 属性には”long_long”と”unsigned_long_long”が許される。
- floatConstant 要素 ― float または double または long double の値を持つ定数を表す。浮動小数点数のリテラルを記述する。type 属性には”float”, ”double”と”long_double”が許される。
- stringConstant 要素 ― 内容にダブルクォーテーションで囲まない文字列を記述する。文字列中の特殊文字は XML（HTML）のルールに従ってクォートされる（’<’は＆lt;に置換されるなど）。type 属性には、”char”と”wchar_t”が許される。C++ではこれらに加えて、"char16_t”と”char32_t”が許される。
  仕様変更：旧仕様では type 属性を持たず、代わりに以下のように定義されている。
  - 属性に is_wide="[1|0|true|false]" (省略時 0)を持ち、1 または true のとき wchar_t 型の文字列を表す。

- moeConstant 要素 － enum 型の定数を表す。内容に enum 定数(列挙型のメンバの名前)を記述する。type 属性は列挙型のタイプ名を記述する。
- booleanConstant 要素 － 真理値リテラル。false または true。type 属性は"bool"のみ許される。
- funcAddr 要素 － 関数のアドレスを表す。内容に関数名を記述する。type 属性は、原則としてその関数のインスタンスの型とするが、翻訳時に不明な場合には別の表現とする。(詳細は実装時に検討する。)

7.2 変数参照の要素 (Var 要素、varAddr 要素、arrayAddr 要素)

変数名への参照を表現する。それぞれ、v, &v, a に対応する(v は配列以外の変数、a は配列変数)。

```
<Var>変数名</Var>
<varAddr>変数名</varAddr>
<arrayAddr>配列変数名</arrayAddr>
```

属性(必須): type, scope

- Var 要素 － 配列以外の変数を参照する式。内容に変数名を指定する。
- varAddr 要素 － 配列以外の変数のアドレスを参照する式。内容に変数名を指定する。
- arrayAddr 要素 － 配列の先頭アドレスを参照する式。内容に配列変数名を指定する。

scope 属性をつかって、局所変数を区別する。

- scope 属性 － "local", "global", "param"のいずれか

例:

n が int 型のとき、n のアドレスの参照 &n は、

```
<varAddr type="P0" scope="local">n</varAddr>
```

と表現される。ここで P0 は、typeTable の中で

```
<pointerType type="P0" ref="int"/>
```

などと宣言されている。

例:

a が int 型の配列のとき、a の参照、すなわち a[0]のアドレスの参照は、

```
<arrayAddr type="A5" scope="local">a</varAddr>
```

と表現される。ここで A5 は、typeTable の中で

```
<arrayType type="A5" element_type="int" array_size="3"/>
```

などと宣言されている。

備考:

a が配列のとき、2015 年 10 月現在の F_Front では &a の参照を a の参照と同様 arrayAddr で表現している。これに関連して Omni XMP では型の不一致によるエラーが出ている(バグレポート 439)。

7.3 pointerRef 要素

式(ポインタ型)の指示先を表現する。

```
<pointerRef>
  式の参照
</pointerRef>
```

属性(必須): type

例:

式 *var1 (var1 は int 型へのポインタ)は以下のように表現される。

```
<pointerRef  type="int">
  <Var type="P0" scope="local">var1</Var>
</pointerRef>
```

要確認:

現状(2015 年 10 月)の C_Front では、*(&var_name) というパターンのとき

<PointerRef><varAddr>var_name</varAddr></PointerRef>

でなく

<Var>var_name</Var>

と表現している。なぜこのパターンに限って簡単化しているのか不明。

7.4 arrayRef 要素

配列要素 a[i]への参照を表現する。

```
<arrayRef>
  arrayAddr 要素
  式の要素
</arrayRef>
```

属性(必須): type

例:

int a[3]; と宣言されているとき、配列要素 a[i] の参照は、

```
<arrayRef type="int">
  <arrayAddr type="A5" scope="local">a</arrayAddr>
  <Var type="int" scope="local">i</Var>
</arrayRef>
```

のように表現される。配列要素のアドレス &a[i] の参照は、

```
<addrOfExpr type="P232">
  <arrayRef type="int">
```

```
    <arrayAddr type="A5" scope="local">a</arrayAddr>
    <Var type="int" scope="local">i</Var>
  </arrayRef>
</addrOfExpr>
```

のように表現される。ここで P232 は int 型へのポインタと宣言されている。後者は arrayAddr 要素でないことに注意されたい。

7.5 メンバの参照の要素（C++拡張）

構造型、クラス、または共用型のオブジェクトを s とするとき、s のメンバ m への参照 s.m、s のメンバ m のアドレスの参照&s.m、s のメンバ配列 a の要素への参照 s.a[i]、および、s のメンバ配列 a の要素のアドレスの参照&s.a[i]を、それぞれ以下のように表現する。

<memberRef> *or* <memberAddr> *or* <memberArrayRef> *or* <memberArrayAddr>
式の要素
</memberRef> *or* </memberAddr> *or* </memberArrayRef> *or* </memberArrayAddr>

属性(必須): type, member

- memberRef － 配列以外のメンバを参照する。member 属性にメンバ名を指定し、子要素でオブジェクトのアドレスを表現する。例えば、オブジェクト s の int 型メンバ n への参照 s.n について、以下のように表現する。

```
<memberRef type="int" member="n">
  <varAddr type="P0" scope="local">s</varAddr>
</memberRef>
```

- memberAddr － 配列名以外のメンバのアドレスを参照する。member 属性にメンバ名を指定し、子要素でオブジェクトのアドレスを表現する。例えば、オブジェクト s の int 型メンバ n のアドレス &s.n について、以下のように表現する。

```
<memberAddr type="int" member="n">
  <varAddr type="P6" scope="local">s</varAddr>
</memberAddr>
```

- memberArrayRef － オブジェクトの配列メンバを参照する。member 属性にメンバ名を指定し、子要素でオブジェクトのアドレスを表現する。例えば、オブジェクト s の int 型配列メンバ a への参照 s.a について、以下のように表現する。

```
<memberArrayRef type="A0" member="a">
  <varAddr type="P1" scope="local">s</varAddr>
</memberArrayRef>
```

- memberArrayAddr － オブジェクトの配列メンバのアドレスを参照する。member 属性にメンバ名を指定し、子要素でオブジェクトのアドレスを表現する。例えば、オブジェクト s の int 型配列メンバ a のアドレス &s.a について、以下のように表現する。

```
<memberArrayAddr type="P24" member="a">
```

```
    <varAddr type="P7" scope="local">s</varAddr>
  </memberArrayAddr>
```

メンバの参照が入れ子になるとき、子要素の表現も入れ子になる。

要検討：構造体まわりの現在の C_Front の変換仕様について

arrayRef 要素（7.4 節）と memberArrayRef 要素、arrayAddr 要素（7.2 節）と memberArrayAddr 要素は、それぞれ名前が似ているが意味の対称性がない。少なくとも名前を再考したい。他の点でも、今後構造体やクラスへの対応を考えると、整理しておきたいところ。

C 言語表現	XcodeML 表現	C 言語表現	XcodeML 表現
v	Var v	s.v	memberRef v varAddr s
&v	varAddr v	&s.v	memberAddr v varAddr s
a	arrayAddr a	s.a	memberArrayRef a varAddr s
&a	arrayAddr a	&s.a	memberArrayAddr a varAddr s
a[i]	arrayRef arrayAddr a Var i	s.a[i]	pointerRef plusExpr memberArrayRef a varAddr s Var i
&a[i]	addrOfExpr arrayRef arrayAddr a Var i	&s.a[i]	plusExpr memverArrayRef a varAddr s Var i

7.6　メンバポインタの参照の要素（C++）

オブジェクト s のメンバへのポインタの参照 s.*p を表現する。

```
<memberPointer>
    式の要素
</memberPointer>
```

属性(必須): type, name

name 属性に変数名を指定し、子要素で構造体のアドレスを表現する。

備考:

メンバの参照(7.5 節)では member 属性にメンバ名を記述するのに対し、メンバポインタの参照(本節)では name 属性に変数名を記述する。この仕様は実装を反映した。

例:

以下のプログラムで、(1)はメンバ変数へのポインタの宣言、(2)はメンバ関数へのポインタの宣言であり、それぞれメンバ変数、メンバ関数をポイントするよう初期化されている(2.1 節の例参照)。(3)の右辺により s1.foo が引数 3 で呼び出され、左辺 s1.data に代入される。

```
struct S {
  int data;
  int foo(int n) { return n + 1; }
};

int S :: *d = &S :: data;                    // (1)
int (S :: *f)(int) = &S :: foo;              // (2)

struct S s1;
s1.*d = (s1.*f)(3);                          // (3)
```

このとき、(3)の左辺は以下のように表現される。

```
<memberPointerRef type="P4" name="d">
  <varAddr type="P3" scope="global">s1</varAddr>
</memberPointerRef>
```

(3)の右辺は以下のように表現される。

```
<functionCall type="int">
  <function>
    <memberPointerRef type="P4" name="f">
      <varAddr type="P3" scope="global">s1</varAddr>
    </memberPointerRef>
  </function>
  <arguments>
    <intConstant type="int">3</intConstant>
  </arguments>
</functionCall>
```

7.7 複合リテラルの要素 (新規)

型 T の複合リテラル (T){ … } および型 T の複合リテラルのアドレス &(T){ … } を表現する。

```
<compoundValue> or <compoundValueAddr>
  value 要素
```

```
</compoundValue> or </compoundValueAddr>
```

属性(必須): type

指示付きの初期化子((T){ [2]=1, .x=2 } のような記述)に対応する表現は持たず、常に展開された表現に変換される(例参照)。

備考:

複合リテラルは、旧仕様書では castExpr 要素で表現すると書かれているが、C_Front の動作と食い違っている。本節は C_Front の動作に合わせて書き起こした。

例:

以下のようなプログラムで、

```
typedef struct { int x, y; } two_int_t;
int n = 20;
foo(&(two_int_t){ 1, n });
goo((two_int_t){ .y=300 });
```

関数 foo の引数は以下の表現となる。

```
<compoundValueAddr type="P6">
  <value>
    <value>
      <intConstant type="int">1</intConstant>
      <Var type="int" scope="local">n</Var>
    </value>
  </value>
</compoundValueAddr>
```

関数 goo の引数は以下の表現となる。

```
<compoundValue type="P6">
  <value>
    <value>
      <intConstant type="int">0</intConstant>
      <intConstant type="int">300</intConstant>
    </value>
  </value>
</compoundValue>
```

7.8 thisExpr 要素 (C++)

thisExpr 要素は、C++の this に対応する。

```
<thisExpr/>
```

属性なし

7.9 assignExpr 要素

assignExpr 要素は、2つの式の要素を sub 要素に持ち、代入を表す。

```
<assignExpr>
  式の要素
  式の要素
</assignExpr>
```

属性(必須): type

属性(optional): is_userDefined

第1の式を左辺、第2の式を右辺とする代入文を表現する。

7.10 2 項演算式の要素

二項演算式を表現する。被演算子の2つの XML 要素を内容に指定する。

```
<二項演算要素名>
  式の要素
  式の要素
</二項演算要素名>
```

属性(必須): type

属性(optional): is_userDefined

二項演算要素名には以下のものがある。

- 算術二項演算子
  - plusExpr － 加算
  - minusExpr － 減算
  - mulExpr － 乗算
  - divExpr － 除算
  - modExpr － 剰余
  - LshiftExpr － 左シフト
  - RshiftExpr － 右シフト
  - bitAndExpr － ビット論理積
  - bitOrExpr － ビット論理和
  - bitXorExpr － ビット論理 排他和
- 代入演算子
  - asgPlusExpr － 加算

- asgMinusExpr － 減算
- asgMulExpr － 乗算
- asgDivExpr － 除算
- asgModExpr － 剰余
- asgLshiftExpr － 左シフト
- asgRshiftExpr － 右シフト
- asgBitAndExpr － ビット論理積
- asgBitOrExpr － ビット論理和
- asgBitXorExpr － ビット論理　排他和

- 論理二項演算子
  - ogEQExpr － 等価
  - logNEQExpr － 非等価
  - logGEExpr － 大なり、または同値
  - logGTExpr － 大なり
  - logLEExpr － 小なり、または等価
  - logLTExpr － 小なり
  - logAndExpr － 論理積
  - logOrExpr － 論理和

備考:

C では代入演算の第1オペランドは必ず lvalue（左辺式）だったが、C++では演算子のオーバーロードがあるためその限りではなくなった。

7.11　単項演算式の要素

単項演算式を表現する。被演算子を内容に指定する。

```
<単項演算要素名>
  式の要素
</単項演算要素名>
```

属性(必須): type

属性（optional）: is_userDefined

単項演算要素名には以下のものがある。

- 算術単項演算子
  - unaryMinusExpr － 符号反転
  - bitNotExpr － ビット反転
- 論理単項演算子
  - logNotExpr － 論理否定
- sizeof 演算子

- sizeOfExpr － 子要素として式の要素または typeName 要素を指定する。
- alignof 演算子(C++)
  - alignOfExpr － 子要素として式の要素または typeName 要素を指定する。
- typeid 演算子(C++)
  - typeidExpr － 子要素として式の要素または typeName 要素を指定する。
- GCC 拡張の演算子
  - gccAlignOfExpr － GCC の__alignof__演算子を表す。子要素に式または typeName 要素を指定する。
  - gccLabelAddr － GCC の&&単項演算子を表す。内容にラベル名を指定する。

7.12 functionCall 要素

functionCall 要素は関数呼び出しを表す。

```
<functionCall>
  <function>
    式の要素
  </function>
  arguments 要素(7.12.1 項)
</functionCall>
```

属性(必須): type

function 要素には呼び出す関数のアドレスを指定する。
arguments 要素には引数の並びを指定する。

7.12.1 arguments 要素

実引数(actual argument)の 0 個以上の並びを表現する。

```
<arguments>
  [ 式の要素
  … ]
</arguments>
```

7.13 commaExpr 要素

コンマ式(第1オペランドと第2オペランドを評価し、第2オペランドの値を返す式)を表す。

```
<commaExpr>
  式の要素
  式の要素
</commaExpr>
```

属性(必須): type

属性(optional): is_userDefined

7.14　インクリメント・デクリメント要素（postIncrExpr, postDecrExpr, preIncrExpr, preDecrExpr）

postIncrExpr 要素、postDecrExpr 要素は、C 言語のポストインクリメント、デクリメント式を表す。preIncrExpr 要素、preDecrExpr 要素は、C 言語のプレインクリメント、デクリメント式を表す。

<postIncrExpr> *or* <postDeclExpr> *or* <preIncrExpr> *or* <preDecrExpr> 　式の要素 </postIncrExpr> *or* </postDeclExpr> *or* </preIncrExpr> *or* </preDecrExpr >

属性(必須): type

属性(optional): is_userDefined

7.15　castExpr 要素（廃止予定）

castExpr 要素は型変換の式(旧仕様)、または複合リテラルを表す。

<castExpr> 　式の要素　*or* value 要素 </castExpr>

属性(必須): type

属性(optional): is_gccExtension

以下の子要素を持つ。

- cast される式、または、複合リテラルのリテラル部

備考:

現在の C_Front では、複合リテラルにこの表現は使われておらず、compoundValue 要素または compoundValueAddr 要素(7.7 節)が使われている。キャストは C++仕様の static_cast, const_cast または reinterpret_cast に変換して表現する方が、バリエーションの削減になるため望ましい。どちらの用途にも使われないのであれば、castExpr は廃止すべきと考える。

7.16　キャスト要素（staticCast, dynamicCast, constCast, reinterpretCast）（C++）

順に、C++の static_cast, dynamic_cast, const_cast および reinterpret_cast を表現する。

<staticCast> *or* <dynamicCast> *or* <constCast> *or* <reinterpretCast> 　式の要素 </staticCast> *or* </dynamicCast> *or* </constCast> *or* </reinterpretCast>

属性(必須): type

C の cast は、staticCast または constCast または reinterpretCast で表現する。

7.17　condExpr 要素

三項演算 x ? y : z を表現する。

```
<condExpr>
  式の要素
  [ 式の要素 ]
  式の要素
</condExpr>
```

属性(必須): type

第2オペランド(2番目の式)は省略されることがある(GNU 拡張対応)。

要確認:

選択される式の type 属性が異なるとき、condExpr 要素の type 属性はどうするべきか？

7.18　gccCompoundExpr 要素

gcc 拡張の複文式に対応する。

```
<gccCompoundExpr>
  compoundStatement 要素
</gccCompoundExpr>
```

属性(必須): type

- compoundStatement 要素　－　複文式の内容を指定する。

7.19　newExpr 要素と newExprArray 要素

new 演算子または new[]演算子から成る式を表現する。

```
<newExpr>
  arguments 要素(7.12.1 項)
</newExpr>
```

属性(必須): type

```
<newArrayExpr>
  式の要素(7 章)
</newArrrayExpr>
```

属性(必須): type

第１と第２の書式は、それぞれ new 演算子と new[]演算子による領域確保を表現する。確保されるデータは、type 属性の型をもつ。第１の書式の子要素は、コンストラクタに渡されるパラメタを表す。

第２の書式の子要素は、確保する配列の長さを表す。

7.20　deleteExpr 要素と deleteArrayExpr 要素

delete 演算子または delete[]演算子から成る式を表現する。

```
<deleteExpr>
  式の要素
</deleteExpr>
```

属性(必須): type

```
<deleteArrayExpr>
  式の要素
</deleteArrayExpr>
```

属性(必須): type

第１と第２の書式は、それぞれ delete 演算子と delete[]演算子による領域解放を表現する。子要素は、解放する領域へのポインタである。

7.21　lambdaExpr 要素

C＋＋のラムダ式を表現する。

```
<lambdaExpr>
  captures 要素
  symbols 要素
  params 要素
  body 要素
</lambdaExpr>
```

属性(必須): type

symbols 要素、params 要素（5.3.4 節）と body 要素は、functionDefinition 要素（5.2 節）の子要素と同様である。

7.21.1 captures 要素

captures 要素は以下の表現である。

```
<captures>
  <byReference>
    [ name 要素
    … ]
  </byReference>
  <byValue>
```

```
    [ name 要素
    … ]
  </byValue>
</captures>
```

属性(optional): default, is_mutable

captures 要素はオプショナルに以下の属性をもつ。

- default 属性 － “by_reference” のとき、スコープデフォルトが参照キャプチャ “[&]” であることを意味し、”by_value”のときデフォルトがコピーキャプチャ “[=]” であることを意味する。省略されたとき、キャプチャがないことを意味する。
- is_mutable 属性 － 1 または true のとき、mutable 指定があることを意味する。0 または false または省略されたとき、mutable 指定がないことを意味する。

子要素の byReference 要素で指定された名前の変数は参照キャプチャされ、byValue 要素で指定された名前の変数はコピーキャプチャされる。それ以外の変数は、default 属性の指定に従う。

8　テンプレート定義要素（C++）

テンプレート定義要素には、以下のものがある。

- structTemplate 要素（8.2 節） － 構造型のテンプレートを定義する。
- classTemplate 要素（8.2 節） － クラステンプレートを定義する。
- funcitionTemplate 要素（8.3 節） － 関数、メンバ関数、演算子オーバーロード、および、ユーザ定義リテラルのテンプレートを定義する。
- aliasTemplate 要素（8.4 節） － 型のエイリアスのテンプレートを定義する。

これらのテンプレート定義要素は、共通して型仮引数を表現する typeParams 要素（8.1 節）をもつ。

8.1　typeParams 要素

テンプレートの型仮引数の並びを指定する。

```
<typeParams>
  [ { typeName 要素（3.2 節）
      [ <value>
        typeName 要素（3.2 節）
      </value> ] }
    … ]
</typeParams>
```

属性なし

以下の子要素をもつことができる。

- typeName 要素　－　型仮引数に対応するデータ型識別名を表現する。
- value 要素　－　子要素として typeName 要素をもつ。関数テンプレートまたはメンバ関数テンプレートにおいて、直前の typeName 要素に対応する仮引数がデフォルト実引数をもつとき、それを表現する。

typeName 要素は、引数の順序で並んでいなければならない。

8.2　structTemplate 要素と classTemplate 要素

データ型定義要素（3 章）の一つ。構造体とクラスのテンプレートをそれぞれ以下のように表現する。

```
<structTemplate>
  symbols 要素（4.3 節）
  typeParams 要素（8.1 節）
  structType 要素（3.9 節）
</structTemplate>
```

属性(optional): lineno, file

```
<classTemplate>
```

```
    symbols 要素(4.3 節)
    typeParams 要素(8.1 節)
    class 要素(3.9 節)
  </classTemplate>
```

属性(optional): lineno, file

以下の子要素をもつ。

- symbols 要素 － 型仮引数に関する id 要素を子要素として持つ。
- typeParams 要素 － 子要素として typeName 要素の並びを持つ。
- structType 要素または class 要素 － 構造体またはクラスの定義

例:

以下のプログラムは、

```
template <typename T>
struct pair { T val1, val2; };
```

以下のように表現される。

```
<structTemplate>
  <symbols>
    <id type="S0" sclass="template_param">
      <name>T</name>
    </id>
  </symbols>
  <typeParams>
    <typeName ref="S0">
  </typeParams>
  <structType type="S1">
    <symbols>
      <id type="S0">
        <name>val1</name>
      </id>
      <id type="S0">
        <name>val2</name>
      </id>
    </symbols>
  </structType>
</structTemplate>
```

ここで、データ型識別名 S0 は typeTable の中で以下のように定義されている。

```
<basicType type="S0" name="any_typename"/>
```

8.3 functionTemplate 要素

関数、メンバ関数、演算子オーバーロード、および、ユーザ定義リテラルのテンプレートを表現する。globalDeclaration 要素(5.1 節)と declaration 要素(5.2 節)の子要素。

```
<functionTemplate>
  symbols 要素(4.3 節)
  typeParams 要素(8.1 節)
  functionDefinition 要素(5.3 節)
</functionTemplate>
```

属性(optional): lineno, file

以下の子要素をもつ。

- symbols 要素 － 型仮引数に関する id 要素を子要素として持つ。
- typeParams 要素 － 子要素として typeName 要素の並びを持つ。
- functionDefinition 要素 － 関数の定義

例：

以下の関数テンプレートについて、

```
template <class T>
T square(const T& x) { return x * x; }
```

型仮引数 T に対するデータ型識別名 X0 と、仮引数 x の型 const T&に対するデータ型識別名 X1 は、typeTable の中で以下のように定義される。

```
<basicType type="X0" name="any_class"/>
<basicType type="X1" is_const="1" is_lvalue="1" name="X0"/>
```

そして、関数テンプレートは以下のように定義される。

```
<functionTemplate>
  <symbols>
    <id type="X0" sclass="template_param">
      <name>T</name>
    </id>
  </symbols>
  <typeParams>
    <typeName ref="X0">
  </typeParams>
  <functionDefinition>
    <name>square</name>
    <symbols>
```

```
        <id type="X1" sclass="param">
          <name>x</name>
        </id>
      </symbols>
      <typeParams>
        <name type="X1">x</name>
      </typeParams>
      <body>
        ・・・(略)・・・
      </body>
    </functionDefinition>
  </functionTemplate>
```

8.4 aliasTemplate 要素

データ型定義要素(3 章)の一つ。using 文による型の別名定義に対するテンプレートを表現する。

```
<aliasTemplate>
  symbols 要素(4.3 節)
  typeParams 要素(8.1 節)
</aliasTemplate>
```

属性(必須): type, name

属性(optional): lineno, file

以下の属性を持つ。

- type － 型の別名に与えられるデータ型識別名
- name － この型の元になる型のデータ型識別名

要検討:

属性名は ref がよいか name がよいか。

以下の子要素を持つ。

- symbols 要素 － 型仮引数に関する id 要素を子要素として持つ。
- typeParams 要素 － 子要素として typeName 要素の並びだけを持つことができる。

定義される別名は、この type 属性値と同じ type 属性値をもつ id 要素で表現され、そのスコープのシンボルテーブル(globalSymbols または symbols)に登録される。

例:

以下の別名テンプレートについて、

```
template <typename T>
using myMap = std::map<int, T&>;
```

型仮引数 T に対するデータ型識別名 X0 と、std::map の第2型引数 T& に対するデータ型識別名 X0 は、typeTable の中で以下のように定義される。

```
<basicType type="X0" name="any_typename"/>
<basicType type="X1" is_lvalue="1" name="X0"/>
```

そして、別名テンプレートは以下のように定義される。ここで、T0 は他で定義されている std::map のデータ型識別名であり、std::map の id 要素の type 属性と一致している。T1 はこの aliasTemplate で定義されたデータ型識別名であり、myMap の id 要素の type 属性と一致している。

```
<aliasTemplate type="T1" name="T0">
  <symbols>
    <id type="X0" sclass="template_param">
      <name>T</name>
    </id>
  </symbols>
  <typeParams>
    <typeName ref="X0">
  </typeParams>
</aliasTemplate>
```

9　テンプレートインスタンス要素（C++）

テンプレートインスタンス要素には、以下のものがある。

- typeInstance 要素（9.2 節） － 構造型、クラス、および型の別名のテンプレートについて、型実引数を与えて具体化する。
- funcitionInstance 要素（9.3 節） － 関数、メンバ関数、演算子オーバーロード、および、ユーザ定義リテラルのテンプレートについて、型実引数を与えて具体化する。

これらのテンプレートインスタンス要素は、共通して型実引数を表現する typeArtuments 要素（9.1 節）をもつ。

9.1　typeArguments 要素

テンプレートのインスタンスの型実引数の並びを指定する。

```
<typeArguments>
  [ typeName 要素（3.2 節）
    … ]
</typeArguments>
```

属性なし

以下の子要素をもつことができる。

- typeName 要素　－　型実引数に対応するデータ型識別名を表現する。

typeName 要素は、引数の順序で並んでいなければならない。

9.2　typeInstance 要素

データ型定義要素（3 章）の一つ。型のテンプレートのインスタンスを表現する。

```
<typeInstance>
  typeArguments 要素（9.1 節）
</typeInstance>
```

属性(optional): type, ref

以下の属性を持つ。

- type 属性 － typeInstance 要素のデータ型識別名、すなわち表現されたインスタンスの型
- ref 属性 － 対応するテンプレートのデータ型識別名

要検討：

属性名は ref がよいか name がよいか。

例：

構造型のテンプレート

```
template <typename T1, typename T2, typename T3>
struct Triple {
   ・・・(略)・・・
};
```

のデータ型識別名を TRI0 とするとき、そのインスタンス

```
triple<int, int*, int**>
```

のデータ型識別名 TRI1 は、以下のように表現される。

```
<typeInstance type="TRI1" ref="TRI0">
   <typeParameters>
      <typeName ref="int">
      <typeName ref="P0">
      <typeName ref="P1">
   </typeParameters>
</typeInstance>
```

ここで、P0 と P1 は、以下のように定義されているデータ型識別名である。

```
      <pointerType type="P0" ref="int" />
      <pointerType type="P2" ref="int" />
      <pointerType type="P1" ref="P2" />
```

9.3 functionInstance 要素

式の要素(7 章)の一つ。関数とメンバ関数のテンプレートのインスタンスを表現する。

```
<functionInstance>
   typeArguments 要素(9.1 節)
   functionCall 要素(7.12 節)
</functionInstance>
```

属性(必須): type, name

型実引数の並びは、スペースで区切られたデータ型識別名で表現する。

例：

functionTemplate 要素(8.3 節)の例において、関数テンプレート

```
template <class T>
T square(const T& x) { return x * x; }
```

の T に対するデータ型識別名は以下の X0、仮引数 x の型 const T&に対するデータ型識別名は以下の X1 である。

```
<basicType type="X0" name="any_class"/>
<basicType type="X1" is_const="1" is_lvalue="1" name="X0"/>
```

ここで、このテンプレート関数の参照

```
square<int>(10)
```

は、以下のように表現される。

```
<functionInstance>
  <typeArguments>
    <typeName ref="int"/>
  </typeArguments>
  <functionCall type="X0">
    <function>
      <funcAddr type="P0">square</funcAddr>
    </function>
    <arguments>
      <intConstant type="init">10</intConstant>
    </arguments>
  </functionCall>
<functionInstance>
```

ここで、funcAddr 要素の type 属性 P0 は、関数 square へのポインタを意味するデータ型識別名である。

備考：

C++対応版作成に当たって再検討していない。

10.1 coArrayType 要素

"#pragma xmp coarray" によって宣言された、Co-Array 型を表す。 次の属性を持つ。

- type － 派生データ型名。
- element_type － Co-Array の要素のデータ型名。データ型名に対応する型が coArrayType のときは、2次元以上の Co-Array 型を表す。
- array_size － Co-Array 次元を表す。

次の子要素を持つ。

- arraySize － Co-Array 次元を表す。arraySize 要素を持つときの array_size 属性の値は "*" とする。

例:

```
int A[10];
#pragma xmp coarray [*][2]::A
```

上記の変数 A の型を表す XML 要素は、次の coArrayType "C2"になる。

```
<arrayType type="A1" element_type="int" array_size="10"/>
<coArrayType type="C1" element_type="A1"/>
<coArrayType type="C2" element_type="C1" array_size="2"/>
```

10.2 coArrayRef 要素

Co-Array 型の変数への参照を表す。
次の子要素を持つ。

- 1 番目の式 － Co-Array 変数を表す式。
- 2 番目以降の式 － Co-Array 次元を表す式。複数の次元を持つ場合は、複数の式を指定する。

10.3 subArrayRef 要素

部分配列の参照を表す。

次の子要素を持つ。子要素を省略することはできない。

- 第一の XML 要素として配列を表す式をもつ。
- 2 番目以降の式 － 添字または添字 3 つ組を表す式。複数の次元を持つ場合は、複数の式を指定する。

10.4 indexRange 要素

3 つ組(triplet)を表す。

次の子要素を持つ。子要素を省略することはできない。

- lowerBound － 下限のインデックスを表す。子要素に式を持つ。
- upperBound － 上限のインデックスを表す。子要素に式を持つ。
- step － インデックスの刻み幅を表す。子要素に式を持つ。

11 その他の要素・属性

11.1

11.2 is_gccExtension 属性

is_gccExtension 属性は、GCC の __extension__ キーワードを XML 要素の先頭に付加するかどうかを定義し、値は 0 または 1 (false または true) である。is_gccExtension 属性は省略可能で、指定しないときは値 0 を指定したときと同じ意味である。次の XML 要素に is_gccExtension 属性を持つことができる。

- id
- functionDefinition
- castExpr
- gccAsmDefinition

例:

"__extension__ typedef long long int64_t" に対応する定義は次のようになる。

```
<id type="long_long" sclass="typedef" is_gccExtension="1">
  <name>int64_t</name>
</id>
```

11.3 gccAsm 要素、gccAsmDefinition 要素、gccAsmStatement 要素

gccAsm 要素・gccAsmDefinition 要素・gccAsmStatement 要素は、GCC の asm/__asm__ キーワードを定義する。子要素として asm の引数の文字列を持つ。

- gccAsm － asm 式を表す。次の子要素を持つ。
- stringConstant (1 個) － アセンブラコードを表す。
- gccAsmDefinition － asm 定義を表す。子要素は gccAsm と同じ。
- gccAsmStatement － asm 文を表す。

次の属性を持つ。

- is_volatile － volatile が指定されているかどうかの情報、0 または 1、false または true。

次の子要素を持つ。

- stringConstant (1 個) － アセンブラコードを表す。
- gccAsmOperands (2 個) － 1 番目が出力オペランド、2 番目が入力オペランドを表す。オペランドを省略する場合は、子要素を持たないタグを記述する。子要素に gccAsmOperand(複数)を持つ。
- gccAsmClobbers (0-1 個) － クロバーを表す。子要素に 0 個以上の stringConstant を持つ。
- gccAsmOperand － 入出力オペランドを表す。

次の属性を持つ。

- match (省略可) － matching constraint の代わりに指定する識別子を表す("[識別子]" に対応)。
- constraint (省略不可) － constraint/constraint modifier を表す。

次の子要素を持つ。

- 式 (1 個) － 入力または出力に指定する式を表す。

例:

```
asm volatile (
        "661:¥n"
        "¥tmovl %0, %1¥n662:¥n"
        ".section .altinstructions,¥"a¥"¥n"
        ".byte %c[feat]¥n"
        ".previous¥n"
        ".section .altinstr_replacement,¥"ax¥"¥n"
        "663:¥n"
        "¥txchgl %0, %1¥n"
        : "=r" (v), "=m" (*addr)
        : [feat] "i" (115), "0" (v), "m" (*addr));
```

```
<gccAsmStatement is_volatile="1">
  <stringConstant><![CDATA[661:¥n¥tmovl .. (省略) ..]]></stringConstant>
  <gccAsmOperands>
    <gccAsmOperand constraint="=r">
      <Var>v</Var>
    </gccAsmOperand>
    <gccAsmOperand constraint="=m">
      <pointerRef><Var>addr</Var></pointerRef>
```

```
        </gccAsmOperand>
      </gccAsmOperands>
      <gccAsmOperands>
        <gccAsmOperand match="feat" constraint="i">
          <intConstant>115</intConstant>
        </gccAsmOperand>
        <gccAsmOperand constraint="m">
          <pointerRef><Var>addr</Var></pointerRef>
        </gccAsmOperand>
      </gccAsmOperands>
    </gccAsmStatement>
```

11.4 gccAttributes 要素

gccAttributes 要素は GCC の __attribute__ キーワードを定義する。子要素として、__attribute__の引数の文字列を持つ。gccAttributes 要素は、gccAttribute 要素を子要素に複数持つ。

- 型を表す XML 要素全てが gccAttributes 要素を子要素に持つ(0～1 個)。
- id 要素が gccAttributes 要素を子要素に持つ(0～1 個)。
- functionDefinition 要素が gccAttributes 要素を子要素に持つ(0～1 個)。

例:
型を表す XML 要素の子要素に、gccAttributes を設定する例

```
typedef __attribute__((aligned(8))) int ia8_t;
ia8_t __attribute__((aligned(16)) n;
```

```
 <typeTable>
    <basicType type="B0" name="int" align="8" size="4"/>
      <gccAttributes>
        <attribute>aligned(8)</attribute>
     </gccAttributes>
    </basicType>
    <basicType type="B1" name="int" align="16" size="4"/>
     <gccAttributes>
        <attribute>aligned(8)</attribute>
        <attribute>aligned(16)</attribute>
      </gccAttributes>
   </basicType>
```

```
  </typeTable>
 <globalSymbols>
   <id type="B0" sclass="typedef_name">
      <name>ia8_t</name>
    </id>
   <id type="B1">
      <name>n</name>
    </id>
 </globalSymbols>
 <globalDeclarations>
    <varDecl>
      <name>n</name>
    </varDecl>
 </globalDeclarations>
```

id 要素、functionDefinition 要素の子要素に、gccAttributes を設定する例

```
void func(void);
void func2(void) __attribute__(alias("func"));


 void __attribute__((noreturn)) func() {
     ...
 }
```

```
<typeTable>
  <functionType type="F0">
    <params>
      <name type="void"/>
    </params>
  </functionType>
  <functionType type="F1">
    <params>
      <name type="void"/>
    </params>
  </functionType>
</typeTable>
<globalSymbols>
```

```
    <id type="F0" sclass="extern_def">
      <name>func</name>
    </id>
    <id type="F1" sclass="extern_def">
      <name>func2</name>
      <gccgccAttributes>
        <gccAttribute>alias("func")</gccAttribute>
      </gccgccAttributes>
    </id>
  </globalSymbols>
  <globalDeclarations>
    <functionDefinition>
      <name>func</name>
      <gccgccAttributes>
        <gccAttribute>noreturn</gccAttribute>
      </gccgccAttributes>
      <body>...</body>
    </functionDefinition>
  </globalDeclarations>
```

11.5 builtin_op 要素

builtin_op 要素はコンパイラ組み込みの関数呼び出しを表す。以下の XML 要素をそれぞれ 0～複数持つ。子要素の順番は関数引数の順番と一致していなければならない。

- 式 － 呼び出す関数の引数として、式を指定する。
- typeName － 呼び出す関数の引数として、型名を指定する。
- gccMemberDesignator － 呼び出す関数の引数として、構造体・共用体のメンバ指示子を指定する。属性に構造体・共用体の派生データ型名を示す ref、メンバ指示子の文字列を示す member を持つ。子要素に配列インデックスを表す式(0-1 個)と、gccMemberDesignator 要素(0-1 個)を持つ。

11.6 is_gccSyntax 属性

is_gccSyntax 属性はそのタグに対応する式、文、宣言が gcc 拡張を使用しているかどうかを定義する。 値として 0 または 1 (false または true) を持つ。この属性は省略可能であり、省略された場合は値に 0 を指定した時と同じ意味になる。

11.7 is_modified 属性

is_modified 属性はそのタグに対応する式、文、宣言がコンパイルの過程で変形されたかどうかを定義する。値として 0 または 1 (false または true) を持つ。この属性は省略可能であり、省略された場合は値に 0 を指

定した時と同じ意味になる。

次の XML 要素に is_gccSyntax 属性、is_modified 属性を持つことができる。

- varDecl
- 文の要素
- 式の要素

12 未検討項目

以下の項目については、本ドキュメントで触れていない。

- 宣言
  - asm (…)
  - 結合指定　extern “C” double x;
- クラス
  - 部分特殊化
  - final
  - 純粋仮想関数、純粋指定子（＝0）
- 例外処理
  - noexcept キーワード
- 属性
  - [[noreturn]] [[carries_dependency]]　[[deprecated]]

13 コード例

例 1:

```
int a[10];
int xyz;
struct {      int x;     int y;} S;
foo() {
      int *p;
      p =   &xyz;            /* 文 1 */
      a[4] = S.y;            /* 文 2 */
}
```

文 1:

```
<exprStatement>
  <assignExpr type=" P6fc98">
    <ponterRef type=" P6fc98">
      <varAddr scope="local" type="P70768">p</varAddr>
    </pointerRef>
    <varAddr type=" P70828">xyz</varAddr>
  </assignExpr>
</exprStatement>
```

もしくは、

```
<exprStatement>
  <assignExpr type=" P6fc98">
   <Var scope="local" type=" P6fc98">p</Var>
     <varAddr type=" P70828">xyz</varAddr>
 </assignExpr>
</exprStatement>
```

文 2:

```
<exprStatement>
 <assignExpr type="int">
```

```
        <pointerRef type="int">
          <plusExpr type=" P6fc98">
            <arrayAddr type=" P708e8">a</arrayAddr>
            <intConstant type="int">4</intConstant>
          </plusExpr>
        </pointerRef>
        <pointerRef type="int">
          <memberAddr type="P0dede" member="y">
             <varAddr type= "P70988">S</varAddr>
          </memberAddr>
      </pointerRef>
    </assignExpr>
  </exprStatement>
```

もしくは、

```
  <exprStatement>
    <assignExpr type="int">
        <pointerRef type="int">
          <plusExpr type=" P6fc98">
            <arrayAddr type=" P708e8">a</arrayAddr>
            <intConstant type="int">4</intConstant>
          </plusExpr>
        </pointerRef>
        <memberRef type="int" member="y">
          <varAddr type= "P70988">S</varAddr>
      </memberRef>
    </assignExpr>
  </exprStatement>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XcodeProgram source="t3.c">
  <!--
    typedef struct complex {
       double real;
       double img;
    } complex_t;

    complex_t x;
    complex_t complex_add(complex_t x, double y);

    main()
    {
       complex_t z;

       x.real = 1.0;
       x.img = 2.0;

       z = complex_add(x,1.0);

       printf("z=(%f,%f)¥n",z.real,z.img);

    }
    complex_t complex_add(complex_t x, double y)
    {
       x.real += y;
       return x;
    }
  -->
  <typeTable>
    <pointerType type="P0" ref="S0"/>
    <pointerType type="P1" ref="S0"/>
    <pointerType type="P2" ref="S0"/>
    <pointerType type="P3" ref="S0"/>
    <pointerType type="P4" ref="S0"/>
    <pointerType type="P5" ref="F0"/>
    <pointerType type="P6" is_restrict="1" ref="char"/>
    <pointerType type="P7" ref="F2"/>
    <structType type="S0">
      <symbols>
        <id type="double">
          <name>real</name>
        </id>
        <id type="double">
          <name>img</name>
        </id>
      </symbols>
    </structType>
    <functionType type="F0" return_type="S0">
      <params>
        <name type="S0">x</name>
        <name type="double">y</name>
      </params>
    </functionType>
    <functionType type="F1" return_type="int">
      <params/>
    </functionType>
```

```
  <functionType type="F2" return_type="int">
    <params/>
  </functionType>
  <functionType type="F3" return_type="S0">
    <params>
      <name type="S0">x</name>
      <name type="double">y</name>
    </params>
  </functionType>
</typeTable>
<globalSymbols>
  <id type="F0" sclass="extern_def">
    <name>complex_add</name>
  </id>
  <id type="S0" sclass="extern_def">
    <name>x</name>
  </id>
  <id type="F1" sclass="extern_def">
    <name>main</name>
  </id>
  <id type="F2" sclass="extern_def">
    <name>printf</name>
  </id>
  <id type="S0" sclass="typedef_name">
    <name>complex_t</name>
  </id>
  <id type="S0" sclass="tagname">
    <name>complex</name>
  </id>
</globalSymbols>
<globalDeclarations>
  <varDecl>
    <name>x</name>
  </varDecl>
  <funcDecl>
    <name>complex_add</name>
  </funcDecl>
  <functionDefinition>
    <name>main</name>
    <symbols>
      <id type="S0" sclass="auto">
        <name>z</name>
      </id>
    </symbols>
    <params/>
    <body>
      <compoundStatement>
        <symbols>
          <id type="S0" sclass="auto">
            <name>z</name>
          </id>
        </symbols>
        <declarations>
          <varDecl>
            <name>z</name>
          </varDecl>
        </declarations>
```

```
      <body>
        <exprStatement>
          <assignExpr type="double">
            <memberRef type="double" member="real">
              <varAddr type="P0" scope="local">x</varAddr>
            </memberRef>
            <floatConstant type="double">1.0</floatConstant>
          </assignExpr>
        </exprStatement>
        <exprStatement>
          <assignExpr type="double">
            <memberRef type="double" member="img">
              <varAddr type="P1" scope="local">x</varAddr>
            </memberRef>
            <floatConstant type="double">2.0</floatConstant>
          </assignExpr>
        </exprStatement>
        <exprStatement>
          <assignExpr type="S0">
            <Var type="S0" scope="local">z</Var>
            <functionCall type="S0">
              <function>
                <funcAddr type="P5">complex_add</funcAddr>
              </function>
              <arguments>
                <Var type="S0" scope="local">x</Var>
                <floatConstant type="double">1.0</floatConstant>
              </arguments>
            </functionCall>
          </assignExpr>
        </exprStatement>
        <exprStatement>
          <functionCall type="int">
            <function>
              <funcAddr type="F2">printf</funcAddr>
            </function>
            <arguments>
              <stringConstant>z=(%f,%f)¥n</stringConstant>
              <memberRef type="double" member="real">
                <varAddr type="P2" scope="local">z</varAddr>
              </memberRef>
              <memberRef type="double" member="img">
                <varAddr type="P3" scope="local">z</varAddr>
              </memberRef>
            </arguments>
          </functionCall>
        </exprStatement>
      </body>
    </compoundStatement>
  </body>
</functionDefinition>
<functionDefinition>
  <name>complex_add</name>
  <symbols>
    <id type="S0" sclass="param">
      <name>x</name>
    </id>
```

```
      <id type="double" sclass="param">
        <name>y</name>
      </id>
    </symbols>
    <params>
      <name type="S0">x</name>
      <name type="double">y</name>
    </params>
    <body>
      <compoundStatement>
        <symbols>
          <id type="S0" sclass="param">
            <name>x</name>
          </id>
          <id type="double" sclass="param">
            <name>y</name>
          </id>
        </symbols>
        <declarations/>
        <body>
          <exprStatement>
            <asgPlusExpr type="double">
              <memberRef type="double" member="real">
                <varAddr type="P4" scope="param">x</varAddr>
              </memberRef>
              <Var type="double" scope="param">y</Var>
            </asgPlusExpr>
          </exprStatement>
          <returnStatement>
            <Var type="S0" scope="param">x</Var>
          </returnStatement>
        </body>
      </compoundStatement>
    </body>
  </functionDefinition>
 </globalDeclarations>
</XcodeProgram>
```