「Omni XcalableMPコンパイラのC++対応内部表現とコード変換器の開発」報告書

株式会社トランス・ニュー・テクノロジー京都研究室

2018年3月30日

内容

[1 本報告書の構成 6](#_Toc510041701)

[2 納品物の構成 7](#_Toc510041702)

[2.1 docs/ ディレクトリ 7](#_Toc510041703)

[2.2 CXXtoXML/ ディレクトリ 8](#_Toc510041704)

[2.3 XcodeMLtoCXX/ ディレクトリ 8](#_Toc510041705)

[2.4 tests/ ディレクトリ 9](#_Toc510041706)

[2.5 schema/ ディレクトリ 10](#_Toc510041707)

[3 ビルド手順 11](#_Toc510041708)

[3.1 Clangのビルド 11](#_Toc510041709)

[3.2 ClangXcodeMLの各ツールのビルド 13](#_Toc510041710)

[3.2.1 CXXtoXMLのビルド 13](#_Toc510041711)

[3.2.2 XcodeMLtoCXXのビルド 13](#_Toc510041712)

[3.3 ビルドされたツールの利用方法 13](#_Toc510041713)

[3.3.1 CXXtoXMLの利用方法 13](#_Toc510041714)

[3.3.2 XcodeMLtoCXXの利用方法 16](#_Toc510041715)

[4 作業報告 17](#_Toc510041716)

[4.1 納品ソフトウェアの内部実装に関する解説 17](#_Toc510041717)

[4.1.1 CXXtoXMLの各ソースコードについて 17](#_Toc510041718)

[4.1.2 XcodeMLtoCXXの各ソースコードについて 18](#_Toc510041719)

[4.2 テスト用ソースコードによる評価 21](#_Toc510041720)

[4.2.1 tests/compile/ サブディレクトリ内のテストについて 22](#_Toc510041721)

[4.2.2 tests/run/ サブディレクトリ内のテストについて 22](#_Toc510041722)

[4.2.2.1 alias\_decl.src.cpp 23](#_Toc510041723)

[4.2.2.2 ambiguous\_member\_access.src.cpp 24](#_Toc510041724)

[4.2.2.3 arithmetic.src.cpp 24](#_Toc510041725)

[4.2.2.4 array.src.cpp 25](#_Toc510041726)

[4.2.2.5 array\_to\_cv\_qual\_type.src.cpp 25](#_Toc510041727)

[4.2.2.6 boolean.src.cpp 26](#_Toc510041728)

[4.2.2.7 cast.src.cpp 27](#_Toc510041729)

[4.2.2.8 class\_derived\_from\_spec.src.cpp 28](#_Toc510041730)

[4.2.2.9 class\_in\_class\_templ.src.cpp 28](#_Toc510041731)

[4.2.2.10 class\_templ\_partial\_spec.src.cpp 29](#_Toc510041732)

[4.2.2.11 class\_templ\_spec.src.cpp 29](#_Toc510041733)

[4.2.2.12 class\_template.src.cpp 30](#_Toc510041734)

[4.2.2.13 comma.src.cpp 31](#_Toc510041735)

[4.2.2.14 conditional.src.cpp 31](#_Toc510041736)

[4.2.2.15 constructor.src.cpp 31](#_Toc510041737)

[4.2.2.16 continue.src.cpp 33](#_Toc510041738)

[4.2.2.17 cpp\_cast.src.cpp 33](#_Toc510041739)

[4.2.2.18 default\_arg.src.cpp 35](#_Toc510041740)

[4.2.2.19 do.src.cpp 35](#_Toc510041741)

[4.2.2.20 enum.src.cpp 36](#_Toc510041742)

[4.2.2.21 for.src.cpp 37](#_Toc510041743)

[4.2.2.22 for\_struct.src.cpp 37](#_Toc510041744)

[4.2.2.23 friend.src.cpp 37](#_Toc510041745)

[4.2.2.24 func\_templ\_expl\_spec\_and\_nontempl.src.cpp 38](#_Toc510041746)

[4.2.2.25 func\_templ\_partial\_spec.src.cpp 39](#_Toc510041747)

[4.2.2.26 functional\_notation.src.cpp 39](#_Toc510041748)

[4.2.2.27 functional\_notation\_with\_nested\_name\_spec.src.cpp 40](#_Toc510041749)

[4.2.2.28 goto.src.cpp 41](#_Toc510041750)

[4.2.2.29 hello\_world.src.cpp 42](#_Toc510041751)

[4.2.2.30 hello\_world\_c.src.c 42](#_Toc510041752)

[4.2.2.31 if.src.cpp 43](#_Toc510041753)

[4.2.2.32 if\_else.src.cpp 43](#_Toc510041754)

[4.2.2.33 if\_stmt\_with\_class\_def.src.cpp 43](#_Toc510041755)

[4.2.2.34 member\_access\_nested\_name\_specifier.src.cpp 44](#_Toc510041756)

[4.2.2.35 namespace.src.cpp 45](#_Toc510041757)

[4.2.2.36 nested\_class\_static\_data\_member.src.cpp 45](#_Toc510041758)

[4.2.2.37 nested\_class\_template.src.cpp 46](#_Toc510041759)

[4.2.2.38 new.src.cpp 47](#_Toc510041760)

[4.2.2.39 new\_array.src.cpp 47](#_Toc510041761)

[4.2.2.40 nullptr.src.cpp 47](#_Toc510041762)

[4.2.2.41 placement\_new.src.cpp 48](#_Toc510041763)

[4.2.2.42 pointer\_to\_member.src.cpp 48](#_Toc510041764)

[4.2.2.43 sizeof.src.cpp 49](#_Toc510041765)

[4.2.2.44 static\_data\_member.src.cpp 49](#_Toc510041766)

[4.2.2.45 storage\_class\_extern.src.cpp 50](#_Toc510041767)

[4.2.2.46 storage\_class\_static.src.cpp 50](#_Toc510041768)

[4.2.2.47 switch.src.cpp 51](#_Toc510041769)

[4.2.2.48 throw.src.cpp 51](#_Toc510041770)

[4.2.2.49 type\_nested\_name\_specifier.src.cpp 51](#_Toc510041771)

[4.2.2.50 unary\_minus.src.cpp 52](#_Toc510041772)

[4.2.2.51 union\_in\_namespace.src.cpp 52](#_Toc510041773)

[4.2.2.52 unnamed\_namespace.src.cpp 53](#_Toc510041774)

[4.2.2.53 using\_decl\_access\_spec.src.cpp 53](#_Toc510041775)

[4.2.2.54 using\_directive.src.cpp 54](#_Toc510041776)

[4.2.3 tests/CCTest/ サブディレクトリ内のテストについて 54](#_Toc510041777)

[4.2.3.1 基本的なテスト 56](#_Toc510041778)

[4.2.3.2 その他のテスト 66](#_Toc510041779)

[4.2.4 テスト結果 74](#_Toc510041780)

# 本報告書の構成

本報告書は、本案件に関する解説書を兼ねた文書であり、下記の内容からなる。

* 納品されたソフトウェア（ClangXcodeML）の利用方法について  
  下記の各パートからなる。
  + gitリポジトリの構成
  + ビルド方法
  + ビルドされたツールの利用方法
* 作業報告書一式  
  下記の各パートからなる。
  + 納品ソフトウェアの内部実装に関する解説
  + テスト用ソースコードによる評価

# 納品物の構成

本案件で開発されたソフトウェアは、GitHub上に設置されたgitレポジトリでバージョン管理されており、納品物はここに置かれている。

gitリポジトリには下記のディレクトリが置かれている。

* docs/ ディレクトリ
* CXXtoXML/ ディレクトリ
* CXXtoXcodeML ディレクトリ
* XcodeMLtoCXX/ ディレクトリ
* tests/ ディレクトリ
* scripts/ ディレクトリ
* schema/ ディレクトリ

## docs/ ディレクトリ

* CtoXcodeMLreport-20180330.docx

本報告書である。

* CXXtoXML-src.md

新しい正変換ツールであるCXXtoXML/src ファイルについての説明であり、

本報告書の4.1.1節の内容として取り込まれている。

* XcodeMLtoCXX-src.md

逆変換ツールであるXcodeMLtoCXXのファイルについての説明であり、

本報告書の4.1.2節の内容として取り込まれている。

* ClangXML.md

現在の正変換ツールが生成するXMLの仕様を説明する文書である。

* doxygen/ サブディレクトリ

空ディレクトリである。

CtoXcodeML/src/および XcodeMLtoCXX/ 内でdoxygenを実行したときに、

このディレクトリ内にドキュメントが生成される。

* 上記の他、過去のドキュメントがas-isで含まれている。

## CXXtoXML/ ディレクトリ

前回のプロジェクトで作ったCXXtoXMLのコードそのものである。現在は使っていない。

* Makefile  
  下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make all と make clean に対応 (make check にも対応しているが、 tests/ サブディレクトリにテストが置かれていないので、 make checkは特に何もしない)。
* src/ サブディレクトリ  
  CXXtoXMLのソースコード一式を収めている。ここにある Makefile を用いてビルドできる。手順は3.2.1節で解説する。
* tests/ サブディレクトリ  
  CXXtoXML単独のテストを配置するためのディレクトリだが、現状ではそのようなテストを作成していない（Makefileのみ存在するが、何もしない）。

## XcodeMLtoCXX/ ディレクトリ

逆変換（XcodeMLを入力にとり、対応する C/C++ ソースコードを出力する操作）ツールである XcodeMLtoCXXに関するファイル一式を収めている。下記のファイル・サブディレクトリからなる。

* Makefile  
  下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make all, make clean と make check にのみ対応。
* src/ サブディレクトリ  
  XcodeMLtoCXXのソースコード一式を収めている。ここにある Makefile を用いてビルドできる。手順は3.2.2節で解説する。
* tests/ サブディレクトリ  
  XcodeMLtoCXX単独での動作テストを収めている。このテストについての報告は前期の報告書(CtoXcodeMLreport-20160929.docx) の4.3.1節に収められている。

## tests/ ディレクトリ

正変換と逆変換の両方を用いた一連のテストをおこなうためのファイル群が置かれている。下記のファイル・サブディレクトリからなる。

* 下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make clean と make check にのみ対応。
* CCTest/ サブディレクトリ  
  本報告書での評価作業に用いたファイル群を収めている。テスト用ソースコード（C++ベンチマーク）はライセンスの都合上この中には含まれていないが、それらのソースコードを含む CppTest.tar.gzというファイルを所定の位置に配置しておくことにより、テストが実施される。テスト手順およびテスト結果については節で解説する。
* compile/ サブディレクトリ  
  正変換・逆変換ツールの動作テスト用のC/C++ソースコードと、それらに対して正変換→逆変換を通した結果を作成してそれぞれ通常のC/C++コンパイラでコンパイルできることを確認するテストを収めている。前期の報告書(CtoXcodeMLreport-20160929.docx)の4.3.2節以降に収めたものに相当するが、正変換としてCXXtoXMLを用いるよう変更してある。
* run/ サブディレクトリ  
  正変換・逆変換ツールの動作テスト用C/C++ソースコードを収めている。

## schema/ ディレクトリ

XcodeML の仕様を XML schema の記法で表したXcodeML\_CXX.xsdファイルを収めている。

# ビルド手順

## Clangのビルド

まず、Clangをソースコードからビルドし、 /usr/local 以下にインストールしておく必要がある。そのあと、ソースコードをビルドする。本開発の開始時点でベースとした Clangのバージョンは9.0.1である。

下記に Clang のインストール手順を述べる。

Clangのビルドには、同バージョンのLLVMコアとcompiler-rtが必要である。いずれも<http://llvm.org/releases/download.html>から各バージョンのものが取得できる。また同じページの先頭のリンクでGitHub から取得することが出来る。

% cd 作業用ディレクトリ  
% git clone https://github.com/llvm/llvm-project  
% cd llvm  
% git checkout llvmorg-9.0.1

% cd 作業用ディレクトリ  
% mkdir build  
 ※llvmのビルド用のディレクトリを準備する。名前は何でもよい。  
% cmake –G Ninja ../llvm  
 ビルド用のサブディレクトリ・ファイル群が生成される。  
　　また、この時点ですでにCコンパイラとしてclangを用いることができる  
　　のであればCC=clang CXX=clang++ という環境変数を設定してcmakeを  
　　おこなってもよい（gcc/g++を用いるよりも高速にビルドが完了する）。  
% ninja  
 ※（PATH環境変数で探せる位置に無いなら適宜フルパスで指定する）  
 ビルドが実行される。マルチコアの場合には自動的に並列ビルドとなる。  
 初回は4コア2スレッドのCPU（8並列）でも二時間ぐらいかかる。  
 明示的に並列度を指定する場合は-jオプションを用いる。ビルドの終盤  
　　でリンクをおこなう際にはメモリ不足になりやすいので、ビルドに失敗  
　　した場合には並列度を下げてやり直すと良い。また、前述のようにして  
　　clang/clang++を用いた方がメモリ消費も少なくて済むようである。  
　　一度ビルドが成功したあとは、再ビルドする際にcmakeから実行する  
　　必要はなく、ninjaコマンドのみを実行すればよい。

以上は通常のRelease版のビルド手順である。これ以外に、デバッグ版のビルドを下記のようにしておこなうことができる。このようにすると、LLVMの各種のコマンド(optなど)で –debug や –debug-only オプションが使えるようになり、コンパイラ内部のデバッグに役立つ情報が得られるようになる。

% cmake –G ninja –DCMAKE\_BUILD\_TYPE=Debug ../llvm  
% ninja  
 ※ビルドを実行（既に実行済みであれば、必要な部分だけリビルド）する。  
 Releaseビルドに比べてさらに多くのメモリを必要とするので、  
 メモリ不足で失敗するようなら-jオプションで並列度を下げて調整する。  
% ninja install  
 ※/usr/local以下にインストールされる。

CMakeのこの他のオプション指定については<http://llvm.org/docs/CMake.html> に書かれているので、そちらを参照されたい。

## ClangXcodeMLの各ツールのビルド

### CXXtoXcodeMLのビルド

ビルドのためにlibxml2が必要である。Ubuntuであれば  
 % sudo apt-get libxml2-dev  
のようにしてインストールする。

gitレポジトリから取得した CXXtoXcodeML/src/ ディレクトリ内で make を実行すればビルドが実行され、 CXXtoXMLという名の実行ファイルが生成される。

make cleanをすると、ビルド結果が全て消される。

### XcodeMLtoCXXのビルド

ビルドのためにlibxml2が必要である。Ubuntuであれば  
 % sudo apt-get libxml2-dev  
のようにしてインストールする。

gitレポジトリから取得した XcodeMLtoCXX/src/ ディレクトリ内で make を実行すればビルドが実行され、 XcodeMLtoCXX/ ディレクトリ直下にXcodeMLtoCXXという名の実行ファイルが生成される。

make cleanをすると、ビルド結果がすべて消される。

## ビルドされたツールの利用方法

### CXXtoXcodeMLの利用方法

例として、

clang -I some-include-dir/ -D user\_defined\_macro=value foo.c

としてコンパイルするような foo.c を対象にする場合、

./CXXtoXML foo.c -- -I some-include-dir/ -D user\_defined\_macro=value

のように「-- の前に対象ファイル名、 -- のあとにその他のオプション」として指定する。その他のオプションが不要なときも「--」が必要である（この -- が無いと ./compile\_commands.json というファイルが用いられる。これについての詳細は<http://clang.llvm.org/docs/JSONCompilationDatabase.html>を参照のこと）。

以下は、 -- よりも左側に指定することのできるCtoXcodeML独自のオプションについて解説する。

./CXXtoXML -help

でオプションの一覧が出る。

-disable-typeTable - disable <typeTable>

-disable-declarations - disable <globalDeclarations>, <declarations>

…これらはそれぞれ、 XMLの一部の出力を抑制するためのオプションである。主にデバッグ用に用いる。

-trace-typeTable - emit traces on <typeTable>

-fulltrace-typeTable - emit full-traces on <typeTable>

-trace-declarations - emit traces on <globalDeclarations>, <declarations>

-trace-rav - trace Recursive AST Visitor

…これらはそれぞれ、 XMLの出力中にコメントとしてデバッグ情報を含めるためのオプションである。

-typenamemap=<string> - a map file of typename substitution

…これはXcodeMLのtypeTableで扱われる型名を置換するためのオプションである。置換したい型名と置換後の型名をそれぞれ空白区切りで交互に並べたファイルを準備し、そのファイル名を上記の<string>のところに指定する。ただしファイルの末尾に空白や改行があると正しく読み込めなくなるので、ファイルの最終行は改行無しで終わらなければならない。

これ以外にもいくつかのオプションがあるが動作保証はされない。

このツールは、ClangASTの情報をなるべくそのまま出力することを意図して作成されているため、出力結果を読むためにはClangASTの内部構造の知識が必要となるが、ClangASTそのものの詳細についてはここでは解説せず、ClangASTがどのようなXMLになるのかの対応関係のみを概説する。

* 全体は Program というXML要素となり、その子要素として clangASTという要素が生成され、そこにClangASTの構造が生成される。
  + clangのDeclクラスはclangDeclという要素で表現され、ノードの種類はclass属性に入り、種類ごとの細かい情報も属性値として表現される。
  + clangのStmtクラスはclangStmtという要素で表現され、ノードの種類はclass属性に入り、種類ごとの細かい情報も属性値として表現される。
  + clangのTypeLocクラスはclangTypeLocという要素で表現され、ノードの種類はclass属性に入り、xcodemLTypeTable（後述）でのデータ型識別名がtype属性として入る。
  + clangのDeclarationNameInfoクラスはclangDeclarationNameInfoという要素で表現され、ノードの種類はclass要素に入り、名前そのものはその要素のコンテンツとして表現される。
  + clangのNestedNameSpecifierクラスはclangNestedNameSpecifierという要素で表現され、xcodemlNnsTable（後述）でのNNS識別名がnns属性として入り、その他の各種の情報が属性値として表現される。
  + clangのConstructorInitializerクラスはclangConstructorInitializerという要素で表現され、各種の情報が属性値として表現される。
* ClangASTの木構造そのものではないが、XcodeML処理系とのやりとりの際に必要となる情報として、xcodemlTypeTableとxcodemlNnsTableをそれぞれ出力する。
  + clangDecl の子要素としてxcodemlTypeTableが生成される。このうち、トップレベルのclangDeclの子要素であるものがXcodeMLにおけるtypeTableと対応する。それ以外の場所にあるものはローカルなtypeTableなどとして扱われる。
  + clangDecl の子要素としてxcodemlNnsTableが生成される。

### XcodeMLtoCXXの利用方法

XcodeMLtoCXXコマンドにはコマンドラインオプションは無く、入力ファイル（XcodeMLファイル）の名前を一つだけ指定する。結果は標準出力に出される。

# 作業報告

## 納品ソフトウェアの内部実装に関する解説

### CXXtoXMLの各ソースコードについて

* XMLVisitorBase.h, XMLVisitorBase.cpp  
  DeclarationsVisitor.cpp, DeclarationsVisitor.h の実装の基底クラスである class XMLVisitorBase を準備している部分。 意味的にはこのさらに上位に CRTPパターンで書かれた RecursiveASTvisitorクラスを基底に持つ。
* XcodeMlNameElem.h, XcodeMlNameElem.cpp  
  clang の AST から XML の<name>要素を出力する部分。
* DeclarationsVisitor.h, DeclarationsVisitor.cpp  
  clang の AST からそれに近い形式のXML要素を生成する部分。 この部分がCXXtoXMLの中でもっとも大きな部分を占める。
* Hash.h  
  clang AST のQualTypeを C++ 標準ライブラリのstd::hashで用いるための テンプレートを定義する。
* TypeTableInfo.h, TypeTableInfo.cpp  
  clang AST のQualTypeで示された値 (型の種別情報) と XcodeML のデータ型識別名との対応関係を管理する部分。 また、必要に応じて前述した<xcodemlTypeTable>要素を生成する。
* InheritanceInfo.h, InheritanceInfo.cpp  
  C++のクラスの派生関係・継承関係の情報を扱う部分。
* NnsTableInfo.h, NnsTableInfo.cpp  
  Clang AST のNestedNameSpecifierで示された値 (nested-name-specifierの種別情報) と XcodeML の NNS識別名との対応関係を管理する部分。 また、必要に応じて前述した<xcodemlNnsTable>要素を生成する。
* ClangOperator.h, ClangOperator.cpp  
  Clang で定義された演算子の種類を XcodeML での演算子名に変換するための ユーティリティ関数を定義する。
* ClangUtil.h, ClangUtil.cpp  
  Clang で定義された各種のオブジェクトを CXXtoXML で用いるための ユーティリティ関数を定義する。
* CXXtoXML.h, CXXtoXML.cpp  
  main関数部分。 与えられたコマンドラインから AST を構成し、 DeclarationsVisitorにAST を渡す部分。

### XcodeMLtoCXXの各ソースコードについて

* LibXMLUtil.h, LibXMLUtil.cpp  
  libxml を用いて XcodeML を容易に解析するためのユーティリティライブラリ。
* XMLString.h, XMLString.cpp  
  libxml の文字列(xmlChar\*)を C++で容易に扱うための ラッパーを定義している部分。
* Stream.h, Stream.cpp  
  CXXCodeGen::Streamクラスを定義している部分。 CXXCodeGen::Streamは、C/C++プログラムを出力するのに便利なストリームのクラスである。
* StringTree.h, StringTree.cpp  
  CXXCodeGen::StringTreeクラスを定義している部分。 CXXCodeGen::StringTreeは、文字列の高速な連接機能を提供する。
* SourceInfo.h, SourceInfo.cpp  
  SourceInfoクラスを定義している部分。 入力された XML の構造すべてにアクセスできる XPath コンテキスト情報と、 そこから解析されたXcodeMl::TypeTable情報(後述)・XcodeMl::NnsTable情報(後述) を 束ねたデータ構造である。
* Util.h  
  便利なテンプレートを集めたヘッダーファイル。
* XcodeMlNns.h, XcodeMlNns.cpp  
  XcoedMl::Nnsクラスを定義している部分。 XcodeMl::Nnsは、(XcodeML/C++の定義する)NNSを表現する。
* XcodeMlName.h, XcodeMlName.cpp  
  XcodeMl::Nameクラスを定義している部分。 XcodeMl::Nameは、識別子、変換関数ID、コンストラクター名など、 C++が定義する各種の名前を表現する。
* XcodeMlOperator.h, XcodeMlOperator.cpp  
  XcodeMLが定義する演算子名とC++の演算子記号の対応関係を取り扱う部分。
* XcodeMlType.h, XcodeMlType.cpp  
  XcodeMl::Typeクラスを定義している部分。 XcodeMl::Typeは、後述するXcodeMl::TypeTableと合わせて XcodeML で定義されるデータ型を表現する。
* XcodeMlTypeTable.h, XcodeMlTypeTable.cpp  
  XcodeMl::TypeTableクラスを定義している部分。 XcodeMl::TypeTableは、データ型識別名と実際のデータ型との 対応関係に関する情報を保存している。
* XcodeMlUtil.h, XcodeMlUtil.cpp  
  XcodeMLの定義する構造に従ったXMLを解析するために便利な関数を集めたファイル。
* XMLWalker.h  
  XMLWalkerクラステンプレートを定義しているヘッダーファイル。 XML の各要素を処理する際、 要素の種類に合わせて別々の処理を行うことが必要になる場合がある。 XMLWalkerはこれを実現する。 後述する NnsAnalyzer、TypeAnalyzer、CodeBuilder で XML の各部分を処理するために使われている。
* AttrProc.h  
  AttrProcクラステンプレートを定義しているヘッダーファイル。 AttrProcを使うことで、与えられた XML の各要素に対し、 その属性(XML attribute)に応じた柔軟な処理を行うことができる。 後述するClangDeclHandler、ClangNestedNameSpecHandler、ClangStmtHandler、 ClangTypeLocHandlerで、 XMLの各要素を処理する ために使われている。
* TypeAnalyzer.h, TypeAnalyzer.cpp  
  XML の<xcodemlTypeTable>部を解析して データ型識別名と実際のデータ型との対応関係を管理する部分。
* NnsAnalyzer.h, NnsAnalyzer.cpp  
  XML の<xcodemlNnsTable>部を解析して NNS識別名と実際のNNSとの対応関係を管理する部分。
* ClangDeclHandler.h, ClangDeclHandler.cpp  
  入力されたXML文書に含まれる<clangDecl>要素を解析して C/C++プログラムを出力する部分。
* ClangNestedNameSpecHandler.h, ClangNestedNameSpecHandler.cpp  
  入力されたXML文書に含まれる<clangNestedNameSpecifier>要素を解析して C/C++プログラムを出力する部分。
* ClangStmtHandler.h, ClangStmtHandler.cpp  
  入力されたXML文書に含まれる<clangStmt>要素を解析して C/C++プログラムを出力する部分。
* ClangTypeLocHandler.h, ClangTypeLocHandler.cpp  
  入力されたXML文書に含まれる<clangTypeLoc>要素を解析して C/C++プログラムを出力する部分。
* CodeBuilder.h, CodeBuilder.cpp  
  XML の各要素を解析して その要素名に応じて適切な C/C++プログラムを出力する部分。
* XcodeMLtoCXX.cpp  
  main関数部分。 コマンドライン引数として与えられたファイル名が表す XML 文書を読み、 上記各 Walker、Handler を用いて C/C++プログラムを出力する。

## 今回の作業

まず、前回のCXXtoXMLから、llvm/clangのコードベースのバージョンを9.0に変更し、pImplパターンを排除したツールCXXtoXcodeMLを作り、それをCXXtoXMLでの動作と同等まで動作するようにした。その後、vectorを使ったプログラムを作り、そこで洗い出された問題をテストプログラムとし、その動作を確認した。最後はvectorをつかったプログラムを一通り通すことを優先し、テンプレートそのものを動作させることは困難という認識から、テンプレートの特殊化を行ったものを出力し、それを処理するようにした。

テストケースを作って対応したものは

* 型エイリアスのテンプレート
* 非型引数のテンプレート
* メンバ存在確認テンプレート

がある。

また可変長テンプレート等のようにとりあえずの処理を行うようにしたものもある。

また、vector.src.cppから展開したXMLコードはvec.xml,vec.dst.cppにあり、テストが通っていないが、とりあえず含めてある。それによるエラーはerrorsというファイルに置いてある。

## テスト用ソースコードによる評価

今回の作業では、理化学研究所側で準備されたC++ ベンチマークテストから選択された所定のソースコードについて、正変換および逆変換が正しく動作するかの評価を主眼におき、下記のような方針で動作を確認した。

* 正変換の結果を逆変換し、その結果がC/C++コンパイラでコンパイルできるものであるかを確認する。
* tests/compile/ サブディレクトリ以下のものを除くいずれのソースコードも、実行すると何らかの文字列（多くは ok または ngの二文字のみ）を表示するように書かれているので、上記のコンパイル結果を実行した結果を確認する。

テスト用に用いたソースコードは、ライセンス的に公開が不可能であるため、 gitレポジトリには収められていない。かわりに、 ClangXcodeMLリポジトリを取得したディレクトリと同じ位置に CppTest.tar.gz というファイルを配置しておく（つまり、下記の手順のCCTestディレクトリからみると../../../CppTest.tar.gzという形で参照できるようにしておく）と、そのtarファイルを展開してテストを実行できるようにしてある。

### tests/compile/ サブディレクトリ内のテストについて

このテストは正変換・逆変換が適格なC/C++プログラムを出力することを検証する。テストの実行には次の手順を踏む。

1. git レポジトリ tests/compile ディレクトリ内をカレントディレクトリとする。以下の手順はすべてこのディレクトリをカレントディレクトリとした状態で行う。
2. make -C ../CCTest build を実行する。  
   このコマンドは、../../CXXtoXML/src と ../../XcodeMLtoCXX/src で make clean; make を実行する。
3. make check を実行する。  
   これがテスト本体である。カレントディレクトリ（tests/compile/）にある .src.cpp ファイルまたは src.c ファイルに正変換を適用した結果を .cpp.xcodemlファイルまたは .c.xcodeml ファイルに出力する。これらのファイルに逆変換を適用した結果を .dst.cpp ファイルまたは .dst.c ファイルに出力する。最後に、各 .dst.cpp/.dst.cファイルをclang++/clangのシンタックスチェック機能を用いて検査する。出力されたC/C++プログラムが適格であればテストは成功し、そうでなければ失敗する。いずれかのテストケースのいずれかの工程が失敗した場合、すなわち、正変換・逆変換の異常終了が発生した場合にはテスト全体が失敗する。

### tests/run/ サブディレクトリ内のテストについて

このテストは、正変換・逆変換がプログラムの意味を変えずにC/C++プログラムを変換することを検証する。

テストの実行には次の手順を踏む。

1. git レポジトリ tests/run ディレクトリ内をカレントディレクトリとする。以下の手順はすべてこのディレクトリをカレントディレクトリとした状態で行う。
2. make -C ../CCTest build を実行する。  
   このコマンドは、../../CXXtoXML/src と ../../XcodeMLtoCXX/src で make clean; make を実行する。
3. make check を実行する。  
   これがテスト本体である。カレントディレクトリ（tests/run/）にある .src.cpp ファイルまたは src.c ファイルに正変換を適用した結果を .cpp.xcodemlファイルまたは .c.xcodeml ファイルに出力する。これらのファイルに逆変換を適用した結果を .dst.cpp ファイルまたは .dst.c ファイルに出力する。こうして .src.cpp/.src.c ファイルおよび .dst.cpp/.dst.c ファイルが得られる。各ファイルをclang++/clangでコンパイルし、.src.out ファイルおよび .dst.out ファイルとして出力する。これらを実行し、その出力を .src.txt ファイルおよび .dst.txt ファイルとする。最後に、対応する .src.txt ファイルおよび .dst.txt ファイルを比較し、同じであれば成功し、異なっていれば失敗する。いずれかのテストケースのいずれかの工程が失敗した場合、すなわち、正変換・逆変換の異常終了、出力プログラムのコンパイル失敗、または実行ファイルの異常終了が発生した場合にはテスト全体が失敗する。

以下では、このテストに用いた各テストケースについて述べる。

#### alias\_decl.src.cpp

C++11で導入されたエイリアス宣言を扱うテスト。

|  |
| --- |
| #include <stdio.h>  using T = int (\*)(int);  int  func1(int x) {  return x \* x + x + 1;  }  int  func2(int x) {  return 2 \* x;  }  int  func3(int x) {  return x - 1;  }  int  main() {  T a[3];  a[0] = func1;  a[1] = func2;  a[2] = func3;  for (int i = 0; i < 10; ++i) {  printf("%d\n", (a[i % 3])(i));  }  } |

#### ambiguous\_member\_access.src.cpp

派生クラス中の曖昧なデータメンバーへnested-name-specifierを用いてアクセスする式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  class ClassA {  public:  int x;  };  class ClassB {  public:  int x;  };  class ClassC : public ClassA, public ClassB {  public:  int  func() {  return ClassB::x;  }  };  int  main() {  ClassC obj;  obj.ClassA::x = 100;  printf("%d\n", obj.func());  } |

#### arithmetic.src.cpp

整数演算および整数演算子の優先順位の取り扱いを確認するテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  printf("%d\n", 2 \* (3 + 10));  } |

#### array.src.cpp

配列の宣言、初期化、アクセス、更新を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int factor[1000];  int  main() {  factor[0] = factor[1] = 1;  for (int p = 2; p < 1000; ++p) {  if (factor[p]) {  continue;  }  for (int n = 2 \* p; n < 1000; n += p) {  factor[n] = p;  }  }  for (int i = 0; i < 1000; ++i) {  if (!factor[i]) {  printf("%d\n", i);  }  }  const int array10\_ci[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  for (int i = 0; i < 10; ++i) {  printf("%d\n", array10\_ci[i]);  }  } |

#### array\_to\_cv\_qual\_type.src.cpp

CV修飾された要素型をもつ配列を取り扱うテスト。

|  |
| --- |
| #include <stdio.h>  int g\_a4i[4];  const int g\_a4ci[4] = {100, 200, 300, 400};  volatile int g\_a4vi[4];  const volatile int g\_a4cvi[4] = {1, 2, 3, 4};  void  func\_pi(int \*pi) {  for (int k = 0; k < 4; ++k) {  printf("int \*: %d\n", pi[k]);  }  }  void  func\_pci(const int \*pci) {  for (int k = 0; k < 4; ++k) {  printf("const int \*: %d\n", pci[k]);  }  }  void  func\_pvi(volatile int \*pvi) {  for (int k = 0; k < 4; ++k) {  printf("volatile int \*: %d\n", pvi[k]);  }  }  void  func\_pcvi(const volatile int \*pcvi) {  for (int k = 0; k < 4; ++k) {  printf("const volatile int \*: %d\n", pcvi[k]);  }  }  int  main() {  int b\_a4i[4] = {2, 4, 6, 8};  const int b\_a4ci[4] = {};  volatile int b\_a4vi[4] = {1, 10, 100, 1000};  const volatile int b\_a4cvi[4] = {3, 6, 9, 12};  func\_pi(b\_a4i);  func\_pci(b\_a4ci);  func\_pvi(b\_a4vi);  func\_pcvi(b\_a4cvi);  func\_pi(g\_a4i);  func\_pci(g\_a4ci);  func\_pvi(g\_a4vi);  func\_pcvi(g\_a4cvi);  } |

#### boolean.src.cpp

真理値リテラル(true, false)を扱うテスト。

|  |
| --- |
| #include <stdio.h>  void  print\_boolean(bool b) {  if (b) {  printf("TRUE\n");  } else {  printf("FALSE\n");  }  }  int  main() {  bool b = true;  print\_boolean(b);  b = false;  print\_boolean(b);  } |

#### cast.src.cpp

Cスタイルの型変換式を扱うテスト。

|  |
| --- |
| #include <stddef.h>  #include <stdio.h>  void  print\_as\_pint(void \*pv) {  int \*pi = (int \*)pv;  printf("%d\n", \*pi);  }  void  print\_as\_string(size\_t address) {  const char \*s = (const char\*)address;  printf("%s\n", s);  }  int  main() {  int i = 100;  print\_as\_pint(&i);  const char \*s = "const char \*";  print\_as\_string((size\_t)s);  } |

#### class\_derived\_from\_spec.src.cpp

クラステンプレートの特殊化から派生したクラスの宣言とインスタンス化を扱うテスト。

|  |
| --- |
| #include <stdio.h>  template <typename T>  struct ClassA {  T classa\_x;  };  struct ClassB : ClassA<char> {  int classb\_y;  };  int  main() {  ClassB objb;  objb.classa\_x = 'A';  objb.classb\_y = 100;  printf("%c %d\n", objb.classa\_x, objb.classb\_y);  } |

#### class\_in\_class\_templ.src.cpp

クラステンプレート中のクラス宣言、およびそのクラスのインスタンス化を扱うテスト。

|  |
| --- |
| #include <stdio.h>  template <typename T>  class ClassA {  public:  class ClassB {  public:  T x;  T  get() {  return x;  }  };  };  int  main() {  ClassA<int>::ClassB obj;  obj.x = 100;  printf("%d\n", obj.get());  } |

#### class\_templ\_partial\_spec.src.cpp

クラステンプレートの部分特殊化の宣言およびその実体化を扱うテスト。

|  |
| --- |
| #include <stdio.h>  template <typename T>  struct ClassA {  int  func() {  return 100;  }  };  template <typename T>  struct ClassA<T \*> {  int x;  int y;  int  func() {  return x + y;  }  };  int  main() {  ClassA<double> a;  printf("%d\n", a.func());  ClassA<int \*> b;  b.x = 100;  b.y = 100;  printf("%d\n", b.func());  return 0;  } |

#### class\_templ\_spec.src.cpp

クラステンプレートの明示的特殊化、および、それによって定義されたクラスのインスタンス化を扱うテスト。

|  |
| --- |
| #include <stdio.h>  template <typename T>  struct ClassA {  int  func() {  return 100;  }  };  template <>  struct ClassA<int> {  int x;  int y;  int  func() {  return x + y;  }  };  int  main() {  ClassA<double> a;  printf("%d\n", a.func());  ClassA<int> b;  b.x = 100;  b.y = 100;  printf("%d\n", b.func());  return 0;  } |

#### class\_template.src.cpp

クラステンプレートの宣言とその実体化を扱うテスト。

|  |
| --- |
| #include <stdio.h>  template <typename T>  class ClassA {  public:  ClassA(T x) {  data = x;  }  T  get() {  return data;  }  private:  T data;  };  int  main() {  ClassA<int> obj1(100);  printf("%d\n", obj1.get());  } |

#### comma.src.cpp

コンマ演算式、またその部分式の副作用の順序を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  int i = 10;  printf("%d\n", (++i, i++, 30));  } |

#### conditional.src.cpp

三項演算式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  for (int i = 0; i < 10; ++i) {  printf("%d\n", i % 3 ?: -1);  }  } |

#### constructor.src.cpp

コンストラクターに関する各種の構文を扱うテスト。

このテストでは以下の内容を試験する。

* 複数のコンストラクターをもつクラスの宣言を正しく出力できること、および、インスタンス化の際に正しいコンストラクターを選択できるよう式を復元すること。
* コンストラクター初期化子リストを正しく出力すること。ここでは、基本クラスの初期化とデータメンバーの初期化をテストする。
* コンストラクターのinline定義、out-of-line定義を文脈に応じて正しく出力できること。

|  |
| --- |
| #include <stdio.h>  class ClassA {  public:  ClassA() : num(42) {  }  ClassA(int i, int j) : num(i \* j) {  }  ClassA(int i, int j, int k);  int num;  };  ClassA::ClassA(int i, int j, int k) : num(i \* j \* k) {  }  class ClassB : public ClassA {  public:  ClassB() : ClassA(), bnum(100) {  bnum += num;  }  ClassB(int i) : ClassA(i, i), bnum(i) {  bnum += num;  }  int bnum;  };  class ClassC : public ClassA {  public:  ClassC() {  }  };  int  main() {  ClassA a1(10, 20);  printf("%d\n", a1.num);  ClassA a2(10, 20, 30);  printf("%d\n", a2.num);  ClassB b1 = ClassB();  printf("%d\n", b1.bnum);  ClassB b2(100);  printf("%d\n", b2.bnum);  ClassC c1 = ClassC();  printf("%d\n", c1.num);  } |

#### continue.src.cpp

continue文を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int main() {  for (int i = 0; i < 10; ++i) {  if (i % 3) {  continue;  }  printf("%d\n", i);  }  return 0;  } |

#### cpp\_cast.src.cpp

C++で導入された各種のキャスト式を扱うテスト。

* static\_cast
* dynamic\_cast
* const\_cast
* reinterpret\_cast

|  |
| --- |
| #include <stdio.h>  class ClassA {  public:  void  print() {  printf("%d\n", i);  }  virtual void  f() {  }  int i;  };  class ClassB {  public:  void  print() {  printf("%lf\n", d);  }  double d;  };  class ClassC : public ClassA, public ClassB {  public:  void  print() {  printf("%c\n", c);  }  char c;  };  void  print\_as\_c(ClassA \*pa) {  ClassC \*pc = dynamic\_cast<ClassC \*>(pa);  if (pc) {  pc->print();  } else {  printf("pc is null");  }  }  void  print\_const(const ClassC &rcc) {  const\_cast<ClassC &>(rcc).print();  }  void  print\_reinterpret\_cast(short \*ps) {  int \*pi = reinterpret\_cast<int \*>(ps);  printf("%d\n", \*pi);  }  int  main() {  ClassC obj;  obj.i = 100;  obj.d = 3.14;  obj.c = 'c';  static\_cast<ClassA>(obj).print();  static\_cast<ClassB>(obj).print();  print\_as\_c(&obj);  print\_const(obj);  int i = 100;  print\_reinterpret\_cast(reinterpret\_cast<short \*>(&i));  } |

#### default\_arg.src.cpp

デフォルト実引数の宣言とデフォルト実引数を利用した関数呼び出し式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  func(int a, int b = 100) {  return a + b;  }  int  main() {  printf("%d\n", func(10));  return 0;  } |

#### do.src.cpp

do文を扱うテスト。

|  |
| --- |
| #include <stdio.h>  void  func1() {  int i = 1;  int n = 0;  do {  n = n + i;  ++i;  } while (i <= 100);  printf("%d\n", n);  }  void  func2() {  int j = 1;  do  printf("func2, %d\n", j++);  while (j < 10);  }  void  func3() {  int i = 0;  do  struct A {  A(int x) {  printf("func3, %d\n", x \* x);  }  } obj(++i);  while (i < 20);  }  int  main() {  func1();  func2();  func3();  return 0;  } |

#### enum.src.cpp

(unscoped) enum型の宣言、列挙子の宣言と使用、enumの初期化子の指定を扱うテスト。

|  |
| --- |
| #include <stdio.h>  enum COLOR {  RED = 100,  YELLOW,  BLUE,  };  void  print\_color(COLOR c) {  switch (c) {  case RED: printf("RED\n"); break;  case YELLOW: printf("YELLOW\n"); break;  case BLUE: printf("BLUE\n"); break;  }  }  int  main() {  print\_color(RED);  print\_color(YELLOW);  print\_color(BLUE);  } |

#### for.src.cpp

for文およびfor文のinit-statementにおける変数宣言を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  int ans = 0;  for (int i = 1; i <= 100; ++i) {  ans += i;  }  printf("%d\n", ans);  return 0;  } |

#### for\_struct.src.cpp

init-statementや本体においてstructクラスが宣言されたfor文を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  for (struct {  void func() {  printf("%d\n", num);  }  int num;  } x = {0};  x.num < 10;  ++x.num) {  x.func();  }  for (int i = 0; i < 10; ++i)  struct ForBody {  ForBody(int x) {  printf("ForBody%d\n", x);  }  } body(i);  } |

#### friend.src.cpp

friend宣言を扱うテスト。

|  |
| --- |
| #include <stdio.h>  class ClassA {  public:  ClassA(int i) {  num = i;  }  friend int func(const ClassA &a);  private:  int num;  };  int  func(const ClassA &a) {  return a.num \* a.num;  }  int  main() {  ClassA obj(20);  printf("%d\n", func(obj));  return 0;  } |

#### func\_templ\_expl\_spec\_and\_nontempl.src.cpp

関数テンプレートの宣言、明示的特殊化、関数テンプレートと同名の非テンプレート関数の宣言、これらの名前が存在するときのオーバーロード解決を扱うテスト。特に、非テンプレート関数が関数テンプレートに優先されることを試験する。

|  |
| --- |
| #include <stdio.h>  template <typename T>  const char \*  func(T x) {  return "func<T>";  }  template <>  const char \*  func(int x) {  return "func<int>";  }  const char \*  func(int x) {  return "func";  }  int  main() {  printf("%s\n", func('c'));  printf("%s\n", func<int>(10));  printf("%s\n", func(10));  return 0;  } |

#### func\_templ\_partial\_spec.src.cpp

関数テンプレートの部分特殊化の宣言とその実体化を扱うテスト。

|  |
| --- |
| #include <stdio.h>  template<typename T>  int func(T x) {  return 100;  }  template<typename T>  int func(T \*x) {  return 200;  }  int main() {  int i = func('a');  int j = func(&i);  printf("%d,%d\n", i, j);  return 0;  } |

#### functional\_notation.src.cpp

関数形式明示的型変換式（explicit type conversion, function notation）を扱うテスト。

|  |
| --- |
| #include <stdio.h>  class ClassA {  public:  ClassA() {  x = 42;  }  ClassA(int i, int j) {  x = i \* j;  }  ClassA(int y) {  x = y;  }  operator char() {  return 'A';  }  int x;  };  typedef void \*PtrT;  void  print\_as\_pint(PtrT p) {  printf("%d\n", \*(int \*)p);  }  int  main() {  int i = 42;  print\_as\_pint(PtrT(&i));  printf("%u\n", unsigned(-1));  ClassA obj = ClassA(1, 2);  printf("%c\n", char(obj));  obj = ClassA(10, 20);  printf("%d\n", obj.x);  obj = ClassA(100);  printf("%d\n", obj.x);  obj = ClassA();  printf("%d\n", obj.x);  } |

#### functional\_notation\_with\_nested\_name\_spec.src.cpp

より複雑な関数形式明示的型変換式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  namespace NS1 {  class ClassA {  public:  ClassA() {  x = 42;  }  ClassA(int i, int j) {  x = i \* j;  }  ClassA(int y) {  x = y;  }  int x;  };  } // namespace NS1  int  main() {  NS1::ClassA obj = NS1::ClassA(1, 2);  obj = NS1::ClassA(10, 20);  printf("%d\n", obj.x);  obj = NS1::ClassA(100);  printf("%d\n", obj.x);  obj = NS1::ClassA();  printf("%d\n", obj.x);  } |

#### goto.src.cpp

goto文を扱うテスト。

|  |
| --- |
| #include <stdio.h>  void  test\_multiple\_labels() {  int i = 0;  label0:  label1:  label2:  label3:  if (i > 9) {  return;  }  ++i;  switch (i % 4) {  case 0:  printf("zero\n");  goto label0;  case 1:  printf("one\n");  goto label1;  case 2:  printf("two\n");  goto label2;  default:  printf("other\n");  goto label3;  }  }  int  main() {  int n = 25;  cond:  printf("%d\n", n);  if (n == 1) {  goto end;  }  if (n % 2) {  n = n \* 3 + 1;  goto cond;  } else {  n /= 2;  goto cond;  }  end:  printf("END\n");  test\_multiple\_labels();  return 0;  } |

#### hello\_world.src.cpp

基本的なC++プログラムが正しく変換されることを確認するテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  printf("Hello, World!\n");  return 0;  } |

#### hello\_world\_c.src.c

基本的なCプログラムが正しく変換されることを確認するテスト。

|  |
| --- |
| #include <stdio.h>  int main() {  printf("Hello, World!\n");  return 0;  } |

#### if.src.cpp

if文を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  if (1)  printf("a\n");  } |

#### if\_else.src.cpp

else節を含むif文を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int main() {  for (int i = 0; i < 10; ++i) {  if (i % 2)  printf("then\n");  else  printf("else\n");  }  return 0;  } |

#### if\_stmt\_with\_class\_def.src.cpp

then節およびelse節が複合文でないstructクラス宣言文であるようなif文を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  for (int i = 0; i < 10; ++i) {  if (i % 2)  struct ClassThen {  ClassThen() {  printf("then\n");  }  } Then;  else  struct ClassElse {  ClassElse() {  printf("else\n");  }  } Else;  }  return 0;  } |

#### member\_access\_nested\_name\_specifier.src.cpp

nested-name-specifierを利用して、本来であれば名前隠しのためにアクセスできないはずの基本クラスのデータメンバーにアクセスする式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  class A {  public:  int x;  int  get() {  return x;  }  };  class B : public A {  private:  int x;  };  int  main() {  B b;  b.A::x = 10;  printf("%d\n", b.get());  } |

#### namespace.src.cpp

名前空間宣言、名前空間のメンバーとしての関数の宣言、名前空間中で宣言された名前へのアクセスを扱うテスト。

|  |
| --- |
| #include <stdio.h>  void  func() {  printf("A\n");  }  namespace NS1 {  void  func() {  ::func();  printf("B\n");  }  namespace NS2 {  void  func() {  printf("C\n");  }  } // namespace NS2  } // namespace NS1  int  main() {  func();  ::func();  NS1::func();  ::NS1::NS2::func();  } |

#### nested\_class\_static\_data\_member.src.cpp

staticデータメンバーを持ち、かつネストしたクラスの宣言、およびそのstaticデータメンバーの定義を扱うテスト。

|  |
| --- |
| #include <stdio.h>  class Outer {  public:  class Inner {  public:  static int member\_i;  };  };  int Outer::Inner::member\_i;  int  main() {  Outer::Inner::member\_i = 20;  printf("%d\n", Outer::Inner::member\_i);  } |

#### nested\_class\_template.src.cpp

ネストしたクラステンプレート宣言とその実体化を扱うテスト。

|  |
| --- |
| #include <stdio.h>  template <typename T1>  class ClassA {  public:  T1 x;  T1  func\_a() {  return x + x;  }  template <typename T2>  class ClassB {  public:  T2 y;  T2  func\_b(T1 z) {  return y + y;  }  };  };  int  main() {  ClassA<int>::ClassB<char> obj;  printf("%d\n", obj.func\_b(1));  } |

#### new.src.cpp

new式、delete式を扱うテスト。

|  |
| --- |
| class A {  public:  explicit A(int, int) {  }  };  int  main() {  A \*pi = new A(0, 0);  delete pi;  return 0;  } |

#### new\_array.src.cpp

配列new式、配列delete式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  class ClassA {  public:  int member\_i;  };  int  main() {  ClassA \*pa = new ClassA[10];  for (int i = 0; i < 10; ++i) {  pa[i].member\_i = i;  }  for (int i = 0; i < 10; ++i) {  printf("%d\n", pa[i].member\_i);  }  delete[] pa;  return 0;  } |

#### nullptr.src.cpp

nullptrリテラルを扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  int \*pi = nullptr;  if (!pi) {  printf("!nullptr\n");  }  } |

#### placement\_new.src.cpp

配置new式、疑似デストラクター呼び出しを扱うテスト。

|  |
| --- |
| #include <stddef.h>  #include <stdio.h>  struct ClassA {  ClassA() : member\_i(4) {  }  int member\_i;  };  void \*operator new(size\_t size, int i, char c) {  return operator new(size);  }  int  main() {  ClassA \*pv = new (100, 'c') ClassA;  printf("%d\n", pv->member\_i);  pv->~ClassA();  operator delete(pv);  } |

#### pointer\_to\_member.src.cpp

メンバーへのポインター型、メンバーへのポインターの宣言と定義、メンバーへのポインターを用いたデータメンバーアクセス式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  struct ClassA {  int member\_i;  };  struct Outer {  struct Inner {  char member\_c;  };  };  int  main() {  int ClassA::\*mpi\_a = &ClassA::member\_i;  ClassA obja;  obja.member\_i = 20;  printf("%d\n", obja.\*mpi\_a);  char ::Outer::Inner::\*mpc\_oi = &Outer::Inner::member\_c;  Outer::Inner objoi;  objoi.member\_c = '#';  printf("%c\n", objoi.\*mpc\_oi);  } |

#### sizeof.src.cpp

種々のsizeof式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  printf("%zu\n", sizeof(double));  int i = 1;  printf("%zu\n", sizeof(i));  int a3i[100];  printf("%zu\n", sizeof(a3i));  } |

#### static\_data\_member.src.cpp

クラスのstaticデータメンバーの宣言および定義を扱うテスト。

|  |
| --- |
| #include <stdio.h>  class ClassA {  public:  static int static\_x;  static void  static\_memfn() {  printf("%d\n", static\_x);  }  };  int ClassA::static\_x;  int  main() {  ClassA::static\_x = 10;  ClassA::static\_memfn();  ClassA::static\_x = 20;  ClassA::static\_memfn();  } |

#### storage\_class\_extern.src.cpp

extern指定子を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int global\_variable;  int  func() {  return global\_variable;  }  int  main() {  extern int global\_variable;  global\_variable = 100;  printf("%d\n", func());  } |

#### storage\_class\_static.src.cpp

static指定子を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  func() {  static int x = 0;  return ++x;  }  int  main() {  printf("%d\n", func());  printf("%d\n", func());  printf("%d\n", func());  } |

#### switch.src.cpp

switch文を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  int count = 0;  int sum = 0;  for (int i = 0; i < 20; ++i) {  switch (i % 5) {  case 0: ++count;  case 1: printf("i mod 5 < 2\n"); break;  case 2: {  printf("i mod 5 = 2\n");  int k = i \* i;  sum += k;  }  default: printf("i mod 5 > 2\n");  }  }  printf("%d %d\n", count, sum);  } |

#### throw.src.cpp

tryブロック、catch節、throw式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int main() {  try {  throw 1;  } catch (int e) {  printf("errno: %d\n", e);  }  } |

#### type\_nested\_name\_specifier.src.cpp

nested-name-specifierを含む型名を扱うテスト。

|  |
| --- |
| #include <stdio.h>  namespace NS1 {  class ClassA {  public:  ClassA(int i, int j) {  x = i;  y = j;  }  int x;  int y;  };  } // namespace NS1  int  main() {  NS1::ClassA obj(20, 30);  printf("%d, %d\n", obj.x, obj.y);  } |

#### unary\_minus.src.cpp

単項マイナス演算式を扱うテスト。

|  |
| --- |
| #include <stdio.h>  int  main() {  int i = -100;  int x = 10 \* (-i);  printf("%d\n", i + x);  } |

#### union\_in\_namespace.src.cpp

グローバルでない名前空間スコープで宣言されたunion型を扱うテスト。

|  |
| --- |
| #include <stdio.h>  namespace NS1 {  union ClassA {  int member\_i1;  int member\_i2;  };  } // namespace NS1  int  main() {  NS1::ClassA obja;  obja.member\_i1 = 100;  printf("%d\n", obja.member\_i2);  } |

#### unnamed\_namespace.src.cpp

無名名前空間宣言およびそのスコープで宣言されたstructクラスを扱うテスト。

|  |
| --- |
| #include <stdio.h>  namespace {  class StructA {  public:  int member\_i;  };  StructA obja;  } // namespace  int  main() {  obja.member\_i = 10;  printf("%d\n", obja.member\_i);  } |

#### using\_decl\_access\_spec.src.cpp

クラススコープのusing宣言を扱うテスト。

|  |
| --- |
| #include <stdio.h>  class A {  public:  int  get() {  return x;  }  protected:  int x;  };  class B : public A {  public:  using A::x;  };  int  main() {  B b;  b.x = 10;  printf("%d\n", b.get());  } |

#### using\_directive.src.cpp

usingディレクティブを扱うテスト。

|  |
| --- |
| #include <stdio.h>  namespace NS1 {  namespace NS2 {  void  func() {  printf("NS1::func\n");  }  } // namespace NS2  } // namespace NS1  int  main() {  using namespace NS1::NS2;  func();  } |

#### func\_template\_nontype.src.cpp

整数型引数を持つ

#include <stdio.h>

template <typename T, int x>

T funcX(T y)

{

T a[x];

int z;

int i;

for(i = 0 ; i < x; i++){

a[i] = i;

}

for(i = 0, z = 0; i < x; i++){

z += a[i];

}

return z;

}

template <typename T1, typename T2>

T2 funcY(T1 x, T2 y)

{

return x+y;

}

int main()

{

printf("%d\n", funcX<int, 3>(4));

funcY<int, double> (1, 2.0);

return 0;

}

#### exist\_template\_src.cpp

メンバの存在確認のテンプレート

#include <stdio.h>

struct B{

int f(){

printf("HOGE\n");

return 0;

};

};

struct C{

int f()

{

printf("FUGA\n");

return 1;

}

};

template <bool v, typename T1, typename T2>

struct if\_;

template <typename T1, typename T2>

struct if\_<false, T1, T2>{

using type = T2;

};

template <typename T1, typename T2>

struct if\_<true, T1, T2>{

using type = T1;

};

struct true\_type{static constexpr bool value = true;};

struct false\_type{static constexpr bool value = false;};

template <typename T>

struct exist\_f {

private:

template<typename S, int (S::\*)() = &S::f>

static true\_type check(S\*);

static false\_type check(...);

public:

static constexpr bool value = decltype(check((T\*)nullptr))::value;

};

int main()

{

if\_<exist\_f<B>::value, B, C>::type b;

if\_<exist\_f<int>::value, B, C>::type c;

b.f();

c.f();

return 0;

}

#### member\_pointer.src.cpp

メンバポインタのテストプログラム

#include <stdio.h>

struct A{

void foo();

void bar(int);

};

void A::foo()

{

printf("FOO\n");

}

void A::bar(int x)

{

printf("BAR %d\n", x);

}

int main()

{

A a;

void (A::\*foofunc)() = &A::foo;

void (A::\*barfunc)(int) = &A::bar;

(a.\*foofunc)();

(a.\*barfunc)(3);

return 0;

}

#### alias\_template.src.cpp

usingを使ったテンプレート

|  |
| --- |
| #include <stdio.h>  template <typename T, int x>  struct A  {  T a[x];  A()  {  for(int i=0; i < x; i++){  a[i] = 0;  }  }  T foo(int i)  {  return a[i];  }  ;  struct D : public A<int , 3>{  using BASE = A<int, 3> ;    };  template <typename T>  using U3 = A<T, 3>;  using Uint3 = A<int, 3>;  template <typename T>  struct E : public U3<T> {};  struct F: public A<int,4> {};  template <typename T1, typename T2>  struct C{  T1 x;  T2 y;  };  template <int x> struct B{  int u[x];  };  int main()  {  U3<int> u;  A<int ,3> a;  B<3> b;  C<int, double> c;  D d;  E<unsigned int> e;  F f;  f.foo(2);  printf("Size%lu %lu %lu %lu\n",  sizeof(u), sizeof(b) , sizeof(c) ,sizeof(d));    return 0;  } |

#### Vector.cpp

Std::vectorのテストプログラム。テストフレームワークで動作しない。

#include <vector>

#include <stdio.h>

int main()

{

std::vector<int> a;

a.push\_back(1);

printf("%d\n", a[0]);

return 0;

}

### Test/ サブディレクトリ内のテストについて

テストの実行には次の手順を踏む。

1. git レポジトリ tests/CCTest ディレクトリ内をカレントディレクトリとする。以下の手順はすべてこのディレクトリをカレントディレクトリとした状態でおこなう。
2. make buildを実行する。  
   これは、 ../../CXXtoXML/src と ../../XcodeMLtoCXX/src での make clean; make の実行をおこなう。
3. make prepare-cpptestを実行する。  
   これは、 ../../../CppTest.tar.gz をこのディレクトリ内に展開し（この時点でCppTest/ というサブディレクトリが作成されてその中にファイル一式が展開される）、その中から testobjects.txt に記載された名前をファイル名のベース部に持つ.ccファイルを選び、 .src.cpp というファイルにしてこのディレクトリの中に配置する。この際、もともとの.ccファイルのmain関数の戻り値が省略されている部分にintを明示するようにファイルを変換する。
4. make check-cpptestを実行する。  
   これがテスト本体である。Testobjeects.txtに記載された名前をファイル名のベース部に持つ.src.cppファイルを対象にして、まずその名前およびそれに続くカンマを行頭に出力する。正変換をした結果を .xcodemlという拡張子で出力し、それを逆変換した結果を .dst.cpp という拡張子で出力する。ここまで成功した時点で一つ目の「OK,」が同じ行に出力される。この .dst.cppをclang++でコンパイルした結果を .out という拡張子で出力する。ここまで成功した時点で二つ目の「OK,」が同じ行に出力される。最後に、この .outを実行し、その結果が同じ行の続きとして出力される。  
   一つ目のOKが出る前に失敗した場合は、CXXtoXcodeML fail または XcodeMLtoCXX fail というメッセージがOKのかわりに生成されて、その行はそこで終わりになる。  
   一つ目のOKが出た後、二つ目のOKが出る前に失敗した場合は、 Compilation failというメッセージがOKのかわりに生成されて、その行はそこで終わりになる。
5. 上記で標準出力に生成された内容と同じものが result.csv として保存されている。また、もし途中でエラーがでていればその内容はerr.logとして保存されている。

テンプレート展開を明示的に行う前のツールでのテストでは、hello\_c.src.cppとvector.cpp以外は通っていた.

vectorを使ったプログラムからCXXtoXcodeMLで正常にXMLを出力し、そのコードをXcodeMLtoCXXで処理してコードを出すことは出来たが、その戻したコードはCXXで通らなかった。

これは、テンプレートの要素を中心に未実装部分が多い点、テンプレートトの特殊化を処理するにあたって、テンプレートの場合はテンプレートの依存関係を見て特殊化する必要があるがそれが行われていないためである。