ABI テスト スイートの設計

文責：スニル・スリヴァスタヴァ
2014 年 7 月 2 日

本文書では、C++ IA64 ABI テスト スイートの設計について解説します。C++ IA64 ABI テスト スイートは、コンパイラが IA64 ABI 仕様に準拠しているかどうかを確認することにより、リンクの互換性を保証するものです。C++ IA64 ABI テスト スイートは、現在 2 種類のターゲットをサポートしています。ターゲットはこの先、さらに増加することが可能です。本テスト スイートは、およそ 400 個のテスト ファイル上に分散された、100 万個強のユニークなクラスで構成されており、フルセットのテストの場合、シングルスレッドの 3 GHz コンピュータ上では、約 1 時間でテストが完了します。

C++ ABI は、クラス継承、仮想関数、関数のオーバーロードなどの高レベルな C++ の構成要素をどのように低レベルにマップするのかを定義します。IA64 ABI は、C++ ABI の中でも最も一般的に使用されているもので、clang/LLVM および g++ コンパイラの両方で使用されています。現行バージョンの仕様は以下で確認可能です：http://mentorembedded.github.io/cxx-abi/abi.html。

異なるコンパイラでコンパイルした C++ コード、または同一コンパイラの異なるリリースをそれぞれリンクして動作するアプリケーションを作成する場合には、この定義に厳密に準拠することが必要になりますが、「正しさ」より「一貫性」のほうがより重要になる業界もあります。そのため、各種変更点からテスト スイートを守るための必要性が生じます。

概要

ABI テスト スイートは、ABI 仕様の各部に準拠する C++ コードを使用することで、ABI 仕様と照らし合わせてコンパイラの実装をテストします。テストではコンパイラによって生成されたレイアウトが「正しい値」であるかどうかを比較します。しかし、各テストの「正しい値」は IA64 ABI 仕様で指定されている規則と、「基底部規則（ベース ルール）」[1]の両方に基づくものでなければならず、このため、真にマルチターゲットであるテスト スイートを書く際の、最も高いハードルとなります。

ABI テスト スイートは現在、以下の 2 種類の構成[2]をサポートしていますが、これ以外にもサポートを増やすことは可能です。

1. 64-bit x86-64 clang 3.4、LP64
2. 32 bit x86 clang 3.4、ILP32

[1] 「基底部規則」という用語は、基底型およびポインタのサイズとアラインメントを定義する規則を意味します。この規則は、IA64 ABI 仕様の「Base Document」のセクション 1.5 に定義されています。
[2] Ubuntu 12.04 でテスト済みですが、新しい x86 プラットフォームなら、どれでも動作するはずです。

オブジェクト レイアウト テストは C/C++ コードで構成されており、クラス定義から自動的に生成されます。このクラス定義はテスト中のコンパイラによってコンパイルされ、ターゲット コンピュータ上で実行されます。このテストでは、テスト中のコンパイラによって処理されるレイアウトが、期待されるレイアウトと一致するかどうかが検証されます。

次のような手法でテスト スイートの一連の「興味深い」クラスが生成されます。

1. 手動：ABI 仕様を精査するもの。
2. 自動：主要パラメータ[3]を変更することにより、「可能なすべての」クラスを生成。
3. ツール：既存コードのクラス構造[4]の「本質」をとらえ、アロケーション アルゴリズムにとって重要な仕様にある、元のクラスに一致する単純化したクラスを再生成する。

特記事項：

- テストでは LLVM の「lit」フレームワークを使用します。
- テスト ファイルはそれぞれ自己完結型で依存関係はありません（共通インクルード ファイルは除く）。
- それぞれのテスト ファイルは、1 回は C ファイルとして、もう 1 回は C++ ファイル[5]として、合計 2 回コンパイルされ、結果として得られた 2 個のオブジェクト ファイルが共通のユーティリティと合わせてリンクされ、実行可能なテストが生成されます。
- ターゲット間での選択は、ABISELECT(…) マクロで行われます。ターゲットには、それぞれ 1 つの引数があります。例：ABISELECT(a,b)。ここで、「a」と「b」は、ターゲットが 2 台ある実装での、2 つのターゲットについて期待される結果です。
- テスト ファイルは静的コンストラクタのメカニズムを使ってランタイム コードが自動的に呼び出されるようにするもので、実行可能ファイルを作成する目的で、任意の個数のファイルがリンクできます。

トップレベルの Python スクリプトは、テスト スイートを実行するために提供されています：

```
python lp64tools.py test -v
```

このスクリプトは、すべての lit 標準オプションを取ります。たとえば、複数のスレッドを有効化するときは、-j<n> とします。

テスト内容

テストでは、クラスのサイズおよびアラインメント、ビットフィールドを含む全基本クラスとフィールドのオフセット、ビットフィールドの配置、仮想関数テーブル（vtable）および VTT の構造（基本クラスのコンストラクション・デストラクション中の vtable を含む）、および全メンバ関数の名前マングルを検証します。

[3] 主要パラメータには次のようなものが含まれます：微分ツリーのデプスおよび幅、仮想ベースであるか非仮想ベースであるか、仮想関数の存在の有無や番号など。
[4] クラスの微分、仮想関数のオーバーライド パターン、その他微細な要素。
[5] C の部分と C++ の部分は、「`#ifdef __cplusplus`」で分離されます。C++ の部分にはクラス定義とテストが含まれ、C の部分には複数のテーブルが含まれます。このテーブルには vtable や VTT などで期待される内容の説明があります。lit ヘッダーではコンパイル/リンク/実行シーケンスを管理します。

生成されるテスト ファイルには次のようなものが含まれます：

- クラス定義
- vtable を満たすメンバ関数用の空のスタブ
- テストを行うコード
- クラス記述子。これは C 構造体で、クラスとその vtable、VTT、基本クラスなどを説明します。

テスト ファイルの例は、この文書の終わりに提示します。

C および C++ の双方のテスト

これはそもそも C++ABI テスト スイートとして作成されていますが、基本クラスやメンバ関数のない構造体の C レベルのテストも一連のものが含まれています。C レベルの構造体は、C および C++ の両方のモードでテストします。

本テスト スイートのカバレッジ

テスト スイートの大部分は、オブジェクト レイアウトとアロケーションのテストに特化されており、次のようなものがテストされます。

- sizeof() 演算子と __alignof__() 演算子によって報告される、クラスのサイズおよびアラインメント
- クラス型のローカル変数の内部にあるフィールドのアドレスからこのクラス型のローカル変数のアドレスを減算することによる、フィールドのオフセット。
- ビットフィールドの配置は、実行時にビットフィールドを異なる値に設定し、構造体の内部にある値を（それがどこにあるべきかの理解に基づき）見つけることによりテストします。テストされる値は、0、1 および（1<<(bitfield_size-1)）です。
- 基本クラスのアドレスから完全なオブジェクトのアドレスを減算することによる、全基本クラスのオフセット（直接的なものと間接的なものの両方）。複数の微分パスを持つ仮想基本クラスの場合、それぞれの微分パスのオフセットがテストされます。ソースコード上に何度も現れる非仮想基本クラスの場合[6]、有効な C++[7] として書くことができないものを除き、すべてのパスがテストされます。
- 期待されたマングリング名の変数を見つけることにより、vtable 変数の内容をテストし、仕様で指定されたものとその内容を比較します。
- クラス型のオブジェクトにアタッチされた vtable の内容と、オブジェクトの基本クラスにアタッチされた vtable の内容をテストします。テストは、コンストラクタとデス

[6] あいまいな基本クラスではあるものの、ある型変換チェーンを指定することにより一義的に解決できるもの。

[7] 構造体が abcd{int p;}; struct efgh:abcd{int e;};struct xyz:efgh,abcd{int t;}*ptr とした場合、直接型変換 ((abcd*)ptr) が無効であるためにテストできないが ((abcd*)(efgh*)ptr) が有効であるためにテストできるものとした場合。

トラクタが実行される間[8]と、完全なオブジェクトが作成され、デストラクションが開始される前に実行します。

- コンストラクション シーケンスとデストラクション シーケンスの点検。これは厳密には、正当性のテストですが、事実上、ctor と dtor のvtable のテストでも、正当性がチェックされます。
- 期待されたマングリング名の変数を見つけることにより、あれば、VTT の内容もチェックし、仕様で指定されたものとその内容を比較します。
- 全メンバ関数の名前は、期待されるマングリング名を参照することによりチェックされます。仮想メンバ関数のマングリング名は、vtable によってテストされますが、この手順は非仮想メンバ関数のマングリングをテストするのに必要です。
- サンクは vtable からポイントされるため、サンクの存在ならびに名前のマングリングが vtable のテストによってテストされます。サンクにより加えられた調整はサンクのマングリング名の中でエンコードされるため、サンク名は基底部規則[9]に左右されます。
- 名前のマングリング。
- 空のクラス。
- マルチスレッド実装用のものを含む初期化ガード変数。
- RTTI 変数のコンテンツおよび名前のマングリング。
- 純粋仮想関数および削除指定関数。
- ポインター-メンバ間のレイアウト（データおよび関数の双方）。
- 配列クッキー、new[] 演算子ならびに delete[] 演算子。

テスト スイートは一部、単純化の仮定を行います。単純化の仮定には 2 つ重要なものがあり、1 つは構造体とクラスが同値であるとするもの、もう 1 つは、メンバ アクセスがアロケーションを変更しないとするものです。

テスト スイートがカバーしない領域

完全を期すため、このテスト スイートがカバー「しない」領域の項目を羅列しますが、このリストは完全には網羅されていません。

- C++ 言語の機能正当性。
- メンバ関数の呼び出し。仮想または仮想ではないもの。本テスト スイートは、vtable が正しく指定された関数を指すかどうかをチェックするだけのものです。
- スタック上またはデータ セクション内における変数の割り当ては、sizeof() 演算子と __alignof__() 演算子により報告されるクラスのサイズおよびアラインメントのテストを超えるものです。
- 例外処理の挙動。

[8] これは C++ 規則のとおり、フル オブジェクトの vtable とは異なります。
[9] 複数のサンクが同じ効果を持つことが可能な場合もあり、コンパイラはサンクの選択に一致するとは限りません。テスト スイートにはサンク名変更のメカニズムが含まれ、このメカニズムにより一連の同値であるサンクが生成されます。また、テストは、コンパイラによって生成されたサンクが期待されるサンクの同等性のセットのメンバと一致する場合には、テスト結果は「パス」とされます。

- 基本的な呼び出し規約。基底部規則の一部であるとみなされるため（これらは今後追加していく予定です）。

名前マングリングのテスト

自動的に生成されるテストは、単純にマングリング名への参照を行うことにより、名前のマングリングもチェックします。名前のマングリングにエラーがあった場合には、リンク時エラーに発展します。名前のマングリングに関するこれより複雑なテストは手書きしますが、これは LLVM の FileCheck ユーティリティを使用して生成されたアセンブリ ファイル内に正しくマングルされた名前が存在するかどうかが確認されます。

テスト ハーネス

テスト ハーネスでは、LLVM ディストリビューションから「lit」フレームワークを使用します。各テスト ファイルの先頭行数行には、実行するテストに対して実行する必要がある行が含まれます。各行が「パス」ステータスで終了すれば、テストは「パス」となります。

```
// RUN: cxx_compiler -c %s -I "common" cxx_rtti -o %t1.o
// RUN: c_compiler -c %s -I "common" -o %t2.o
// RUN: c_compiler -c -o %t3.o -I "common" "common/testsuite.c"
// RUN: linker -o %t.exe -I "common" %t1.o %t2.o %t3.o
// RUN: runtool %t.exe | checker "TEST PASSED"
```

上記の行内のキーワードは次のように定義されます。

- `cxx_compiler` は、テスト時に C++ コンパイラに伸張されます。
- `c_compiler` は、対応する C コンパイラに伸張します。
- `linker` はリンカ、またはリンクが必要になるコンパイラ コマンドです。
- `common` は、ユーティリティ ファイル testsuite.c と testsuite.h を含むディレクトリです。
- `%t[num]` は、一意の文字列に伸張されます（通常の lit 操作の場合）。
- `runtool` は、ターゲット上でリンクされているプログラムを実行するのに必要なツールです。ネイティブ プラットフォームでの実行を行う際は、空の文字列としてください。
- `checker` は `grep` にデフォルト設定されますが、ユーザーが独自のツールを提供することも可能です。このツールは、エラー メッセージを検討し、それが「期待される」ものであるのかどうかを判断できるものとします。これは拡張性のために行われるもので、テスト スイートにより期待されるレイアウトからわずかに変化したものをハンドルできるようにします。

トップレベルの Python ファイルは、これらのシンボルの値を判断します。当社のテスト スイートでは、LLVM に付属している lit ハーネスを修正なしで利用しています。そのため、テスト スイートに lit のコピーをインクルードする必要はありません。

構成の例

テストの構成は Python によって行われます。トップレベルのファイルは次のように見えます。

```
import os
import sys
test_params = {
    "c_compiler" :"clang" ,
    "cxx_compiler" :"clang -x c++" ,
    "Mode" :"LP64", # 指定可能な値は "ILP32" または "LP64"
    "bindump" :"nm", # オブジェクト ファイルのシンボル名をダンプするツール
    "runtool" :"", # クロス コンパイラである場合
    "Platform" :"x64",
    "cxx_rtti" :"-frtti", # rtti を有効化するオプション
    "cxx_exceptions" :"-fexceptions", # 例外を有効化するオプション
    "cxx_cpp11" :"" # C++11 モードを有効化するオプション。clang では不要
}
builtin_parameters = {
    'build_mode' :"Release",
    'llvm_site_config' : os.path.join(os.getcwd(), 'lit.site.cfg'),
    'clang_site_config': os.path.join(os.getcwd(), 'lit.site.cfg'),
    'test_params' : test_params
    }
lit.main(builtin_parameters)
```

test_params および builtin_parameters は Python のディクショナリです。コンパイラは要素 “c_compiler” および “cxx_compiler” として指定されます。リンカは別個に指定可能ですが、指定しない場合、“c_compiler” がデフォルトで使用されます。

テスト ファイルの例

典型的なテストは以下のようになります。

```
// RUN: c_compiler -c -o %t1.o -I "common" "common/testsuite.c"
// RUN: cxx_compiler cxx_rtti -c %s -I "common" -o %t2.o
// RUN: c_compiler -c %s -I "common" -o %t3.o
// RUN: linker -o %t2.self %t1.o %t2.o %t3.o
// RUN: runtool %t2.self | checker "TEST PASSED"
#include "testsuite.h"
#ifdef __cplusplus
struct efgh { long r;};
static void Test_efgh(){
  init_simple_test("efgh");
  efgh lv;
  check2(sizeof(lv), ABISELECT(8,4), "sizeof(efgh)");
  check2(__alignof__(lv), ABISELECT(8,4), "__alignof__(efgh)");
  check_field_offset(lv, r, 0, "efgh.r");
}
static Arrange_To_Call_Me vefgh(Test_efgh, "efgh", ABISELECT(8,4));
#else // __cplusplus
... some C data structure desribing abcd
#endif // __cplusplus
#ifdef __cplusplus
struct abcd : efgh {
  int pa;
  virtual void foo(); // _ZN4abcd3fooEv
   ~abcd(); // _ZN4abcdD1Ev
   abcd(); // _ZN4abcdC1Ev
};
void abcd ::foo(){vfunc_called(this, "_ZN4abcd3fooEv");}
abcd ::~abcd(){ note_dtor("abcd", this);}
abcd ::abcd(){ note_ctor("abcd", this);}
static void Test_abcd()
{
  extern Class_Descriptor cd_abcd;
  void *lvp;
  ABISELECT(double,int) buf[4];
  init_test(&cd_abcd, buf);
  abcd *dp, &lv = *(dp=new (buf) abcd());
  lvp = (void*)&lv;
  check2(sizeof(lv), ABISELECT(24,12), "sizeof(abcd)");
  check2(__alignof__(lv), ABISELECT(8,4), "__alignof__(abcd)");
  check_base_class_offset(lv, (efgh*), ABISELECT(8,4), "abcd");
  check_field_offset(lv, pa, ABISELECT(16,8), "abcd.pa");
  test_class_info(&lv, &cd_abcd);
  dp->~abcd();
  Check_Ctor_Dtor_Calls(lvp);
}
static Arrange_To_Call_Me vabcd(Test_abcd, "abcd", ABISELECT(24,12));
#else // __cplusplus
.. some C data structure describing abcd, its vtables etc ...
#endif // __cplusplus
```