THE C++ ABI FOR DUMMIES

Louis Dionne <@LouisDionne>

C++ Standard Library Engineer @ Apple

OUTLINE

- What's specified in the ABI?
- ABI vs source changes
- Controlling the ABI
- Useful tools

WHAT IS AN ABI?

Wikipedia:

> *An ABI defines how data structures or computational routines are accessed in machine code, which is a low-level, hardware-dependent format [...]*

[...] in contrast, an API defines this access in source code, which is a relatively high-level, hardware-independent, often human-readable format.

THE ITANIUM C++ SPECIFICATION

https://github.com/itanium-cxx-abi/cxx-abi

CONVENTION FOLLOWED BY VENDORS

DESCRIBES THE ABI FOR C++ CONSTRUCTS

LAYERED ON TOP OF THE UNDERLYING C ABI

A FEW EXAMPLES

PASSING ARGUMENTS IN C

1. Arguments are passed in registers
2. Or via the stack

PASSING ARGUMENTS IN C++

1. Same as C for trivial types
2. For non-trivial types:
    1. Space allocated on the stack
    2. Caller invokes copy-constructor
    3. Address passed as a normal argument
    4. Caller invokes destructor

LAYING OUT BASE CLASSES

- Do base class members come before or after the derived class members?
- Multiple inheritance: left-to-right, right-to-left, something else?

HOW IT WORKS

- Base classes in declaration order
- Non-static data members in declaration order
- Virtual bases in inheritance graph order

THIS IS WHERE EBO TAKES PLACE

NAME MANGLING IN C

```
void foo(int); // just 'foo'
```

NAME MANGLING IN C++

Consider overloading, namespaces, etc.

Linker has to know which one to call

```cpp
namespace hello {
  void foo(int); // mangled as '_ZN5hello3fooEi'
  void foo(long); // mangled as '_ZN5hello3fooEl'
}
```

ABI STABILITY

Software compiled against one version of a library doesn't need to be recompiled in order to use a newer version of the library

EXAMPLE OF BREAKING ABI

LIBRARY VERSION 1

```
template <typename First, typename Second>
struct pair {
  First first;
  Second second;

  pair(First const& f, Second const& s)
    : first(first), second(s)
  { }

  ~pair() { }
};

void foo(pair<int, int>); // defined in a .cpp
```

LIBRARY VERSION 2

```
template <typename First, typename Second>
struct pair {
  First first;
  Second second;

  pair(First const& f, Second const& s)
    : first(first), second(s)
  { }

  ~pair() = default;
};

void foo(pair<int, int>); // defined in a .cpp
```

APPLICATION

```cpp
#include "library.hpp"

int main() {
  pair<int, int> x = {3, 5};
  foo(x);

  // ...
}
```

WHAT'S THE PROBLEM?

- `pair` version 2 is trivial when `First` and `Second` are trivial
- Passed in registers instead of on the stack

EVEN SEEMINGLY INNOCUOUS CHANGES CAN BREAK ABI!

GENERAL GUIDELINES

(non-exhaustive)

Taken from:

- KDE ABI Guidelines: https://bit.ly/2ka1ITz
- Android ABI Stability docs: https://bit.ly/2lIleac

SAFE (1/3)

- add new non-virtual functions
- add a new enum to a class.
- append new enumerators to an existing enum
    - make sure the underlying type doesn't change

SAFE (2/3)

- define an inline function out-of-line
  - it must be OK for the program to call the old OR the new implementation
- remove private non-virtual functions or static members
  - must not have been used by a function in headers

SAFE (3/3)

- add new static data members
- change default arguments to a method
  - existing calls will use the old default arguments until recompiled
- add new classes
- add or remove friend declarations

WHY IS ABI STABILITY IMPORTANT?

- Ship libraries to a system and update them without
  - recompiling other system libraries
  - recompiling all applications
- Allows shipping static archives

COSTS OF ABI STABILITY

1. Harder to evolve language and library
2. Performance improvements may be hindered

CONTROLLING YOUR ABI

SYMBOL VISIBILITY

(for dynamic libraries)

```
#define HIDDEN_VISIBILITY \
  __attribute__((__visibility__("hidden")))

HIDDEN_VISIBILITY void foo();
```

CONTROLLING LINKAGE

(for static libraries)

```
#define INTERNAL_LINKAGE \
  __attribute__((internal_linkage))

INTERNAL_LINKAGE void foo();
```

CONTROLLING VTABLE AND RTTI VISIBILITY

Today

```
// in header
class __attribute__((__type_visibility__("default"))) Widget {
public:
  virtual void draw();
};

// in library
void Widget::draw() { /* ... */ } // vtable and RTTI implicitly here
```

CONTROLLING VTABLE AND RTTI VISIBILITY

Wish (http://wg21.link/p1263)

```
// in header
class Widget {
public:
  virtual void draw();
};

// in library
// control where and how vtable is instantiated
extern __attribute__((__visibility__("default"))) Widget::virtual;
void Widget::draw() { /* ... */ }
```

WILL THESE ATTRIBUTES EVER BE STANDARDIZED?

- People have tried and failed, so far
- Different platforms are too different (Windows/Unix)

TOOLS

LOOKING AT EXPORTED SYMBOLS

(and their type)

```
$ nm -gmU /usr/lib/libc++.dylib
[...]
external __ZNKSt13bad_exception4whatEv
external __ZNKSt13runtime_error4whatEv
external __ZNKSt16nested_exception14rethrow_nestedEv
external __ZNKSt18bad_variant_access4whatEv
external __ZNKSt19bad_optional_access4whatEv
external __ZNKSt3__110__time_put8__do_putEPcRS1_PK2tmcc
external __ZNKSt3__110__time_put8__do_putEPwRS1_PK2tmcc
external __ZNKSt3__110error_code7messageEv
external __ZNKSt3__110moneypunctIcLb0EE11do_groupingEv
external __ZNKSt3__110moneypunctIcLb0EE13do_neg_formatEv
external __ZNKSt3__110moneypunctIcLb0EE13do_pos_formatEv
[...]
```

PROTIP: `c++filt`

Will demangle anything

```
$ nm -gmU /usr/lib/libc++.dylib | c++filt
[...]
external std::bad_exception::what() const
external std::runtime_error::what() const
external std::nested_exception::rethrow_nested() const
external std::bad_variant_access::what() const
external std::bad_optional_access::what() const
external std::__1::__time_put::__do_put(char*, char*&,
                                        tm const*, char, char) const
external std::__1::__time_put::__do_put(wchar_t*, wchar_t*&,
                                        tm const*, char, char) const
external std::__1::error_code::message() const
external std::__1::moneypunct<char, false>::do_grouping() const
external std::__1::moneypunct<char, false>::do_neg_format() const
external std::__1::moneypunct<char, false>::do_pos_format() const
[...]
```

libabigail

```
$ abidiff libtest-v0.so libtest-v1.so
Functions changes summary: 0 Removed, 1 Changed, 0 Added function
Variables changes summary: 0 Removed, 0 Changed, 0 Added variable

1 function with some indirect sub-type change:

  [C]'function void foo(S0*)' has some indirect sub-type changes:
        parameter 0 of type 'S0*' has sub-type changes:
          in pointed to type 'struct S0':
            size changed from 32 to 64 bits
            1 base class insertion:
              struct type_base
            1 data member change:
             'int S0::m0' offset changed from 0 to 32
```

ANDROID ABI CHECKER

EXTRACTS ABI INFORMATION FROM HEADERS

```
$ header-abi-dumper foo.cpp -o foo.sdump
$ header-abi-linker foo.sdump -o libfoo.lsdump
```

ALLOWS DIFFING ABI INFORMATION

```
$ header-abi-diff -old libfoo_old.lsdump \
                  -new libfoo_new.lsdump \
                  -o libfoo.abidiff
```

OUTPUT

```
$ cat libfoo.abidiff
record_type_diffs {
  fields_diff {
    old_field {
      referenced_type: "foo"
      field_offset: 0
      field_name: "mfoo"
      access: public_access
    }
    new_field {
      referenced_type: "foo *"
      field_offset: 0
      field_name: "mfoo"
      access: public_access
    }
  }
}
```

THANK YOU

Louis Dionne <@LouisDionne>

C++ Standard Library Engineer @ Apple