What's an ABI and why is it so complicated?

Jonathan Wakely
Red Hat

Introduction

Once again, I'm talking about whatever it is that I've been hacking on recently.

- What's an ABI?
- Why does it matter?
- How can you manage your code's ABI?
- Really ugly hacks (time permitting)

Caveat: some of this stuff might be more relevant to Unix-like systems. Please correct me if I say something that isn't true for Windows or other systems.

Is that like an API?

You all know what an Application *Programming* Interface is.

The names of functions you can call, their argument types, return values, effects etc.

For example, the Berkeley sockets API, Win32, OpenGL, POSIX.

But knowing the API is not sufficient to use some library you download from the internet, it might not work for your CPU or OS … but why not?

Application *Binary* Interface

Behind an API you also rely on an ABI, usually defined by the OS.

For example, the **System V Application Binary Interface** documents:

> a standard binary interface for application programs on systems that implement an operating system that complies with the X/Open Common Application Environment Specification, Issue 4.2 and the System V Interface Definition, Fourth Edition.
>
> *System V Application Binary Interface*
> *http://www.sco.com/developers/devspecs/gabi41.pdf*

(The System V ABI is used by most Unix-like systems in use today)

Rules for compilers

Unless you're compiling for bare-metal embedded systems where your program image runs directly on the hardware, the compiler needs to follow the rules of the platform.

The OS must recognise your program as a valid executable to be able to run it, and your program needs to know how to call functions in system libraries.

> […] the resulting executable programs use the specified interface to all system routines and services, and have the format described in the ABI specification.
>
> *System V Application Binary Interface*
> *http://www.sco.com/developers/devspecs/gabi41.pdf*

Rules for compilers

An ABI specification should define at least:

- function calling conventions and symbol naming
- representation of data types (size, alignment, floating point format)
- object file format e.g. ELF, PE/COFF, Mach-O
- how programs are loaded, dynamic linking, symbol relocations, thread-local storage

For the System V ABI these specs are split into generic and processor specific parts.

Calling Conventions

Calling conventions dictate how functions are called, specifically which registers are used and how.

- Which direction the stack grows and where the stack pointer is
- Any alignment requirements for the stack
- How arguments are passed:
    - Registers might be used for simple types, so which ones get used in which order must be known
    - Other arguments are placed on the stack, so their positions and order must be known
- Which registers are used for return values
- Which registers are preserved across the call and which are not

Symbol naming

When the compiler generates code to call a function it needs to know how the function's name is represented, so the linker can find it and the right function can be called.

Usually a C function such as `void frob()` is simply called `frob` in the assembly code, but some ABIs (or other languages) might use something different, like `_frob`.

Representation of Types and Values

C data types are mapped to some native representation supported by the CPU

On x86 `short` is a signed 16-bit object ("halfword") and `int` is a signed 32-bit object ("word")

Floating point types typically use IEEE single-, double- or extended-precision formats

Alignment restrictions on types will affect stack layout, and padding in structs

Bit-field layout must be specified

How is a null pointer represented?

So that's an ABI

Armed with that information your compiler and linker can generate valid programs and libraries that can run and call other code on your target system

And then came C++ ...

C++ complicates things **significantly**

There are lots more things that get done implicitly by the language and so look simple in the code, but in order to reuse existing toolchains designed for C those features must be implemented explicitly by the compiler at the binary level

Why is it so complicated?

- Inheritance
- Namespaces and overloading
- Virtual functions
- Exception handling
- RTTI, vague linkage, local statics, new[]
- …

Inheritance

In addition to the rules the compiler must follow for struct layout, which are inherited from the platform ABI used for C programs, a C++ compiler must decide how to arrange the layout for types with base classes

Should base class sub-objects come before or after the derived type's members?

With multiple inheritance, should bases be laid out left to right, or right to left?

Namespaces and overloading

Extern functions in C have unique names, so as we saw earlier the symbol name for `void frob()` can be simply `frob`

In C++ we can have `void blargle()` and `void blargle(int)` and `void argle::blargle()` in the same program

The compiler needs to generate a unique name for each entity with external linkage, so it uses a *name mangling* scheme to encode scope and argument types into symbol names:

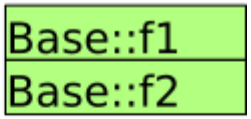
```
_Z7blarglev
_Z7blarglei
_ZN5argle7blargleEv
```

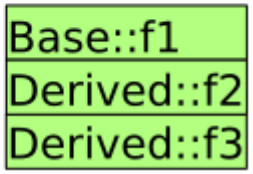
Virtual functions

Each polymorphic class has a *vtable* in the program, which is effectively an array of function pointers

Each object of a polymorphic type contains an extra data member inserted by the compiler, called the *vptr*, which is just a pointer to the vtable

```
struct Base {
  virtual void f1();
  virtual void f2();
  void* __vptr;
};
struct Derived : Base {
  virtual void f2();
  virtual void f3();
};
```





The ABI dictates the layout and use of the vtable and vptr

Exception handling

When you throw and exception in your C++ code the compiler translates that into a call to some function provided by the C++ runtime (called `__cxa_throw` in the Itanium C++ ABI)

The runtime allocates an exception object, looks for a matching `catch (`*`handler`*`)` and starts unwinding the stack

If a handler is found another function in the runtime is called to get access to the exception object that was created, in order to initialize the handler

The ABI dictates the names and semantics of the runtime functions, and how the information about active handlers is represented

RTTI, vague linkage, local statics, `new[]`

The compiler generates `std::typeinfo` objects for the types in the program, and the information needed for `dynamic_cast` to work

Template instantiations, inline functions and vtables might all be emitted in multiple translation units, so the compiler and linker need to have a mechanism for merging or discarding duplicate copies (e.g. COMDAT sections)

Local static variables need thread-safe initialization, which needs to be done consistently across different translation units

Array `new[]` needs to store the number of elements allocated somewhere, so the correct number of destructors can be run by `delete[]`

What isn't part of the ABI?

Things with internal linkage, such as namespace-scope functions declared `static` or in an unnamed namespace

Whether a member function is `public` or `private` only affects whether a call to it compiles or not, it shouldn't affect the binary output

Similarly for deleted functions, since you can't call one, it can't have any effect on the compiled code

Default arguments of functions

Itanium C++ ABI

So when using C++ there are many more things that the implementation has to choose how to represent

There is some sanity though, there's a cross-vendor C++ ABI specification (originally developed for Itanium compilers but used on any processors now) which defines how C++ compilers should do name mangling and exception handling: http://mentorembedded.github.io/cxx-abi/abi.html

GCC and Clang and some other compilers follow that specification, so code compiled with GCC and Clang can be linked together and interoperate.

Why is it even more complicated?

- Namespaces and overloading
- Virtual functions
- Exception handling
- RTTI, vague linkage, local statics, new[]
- **Standard Library implementation**

Standard Library ABI

For C programs there are a number of structs such as `div_t` (and POSIX types like `stat` and `timeval`) that must have a fixed layout for a given ABI

C++ adds many more types in its standard library, all of which can be affected by the ABI complications we've been looking at

Inline functions and templates expose implementation details to users, and private members and the order in which virtual functions are declared is also visible in headers

This means that almost everything in the Standard Library has to be considered as part of the implementation's ABI!

Why does this matter?

The ABI is what allows you to link to other code, either other files in your own project or third-party libraries

Your toolchain + OS define an ABI, but any C++ library defines its own ABI too, which is affected by:

- The functions that define the public API, which will be compiled to some mangled name that the linker uses
- The types which use expose their details such as the order of data members, and any virtual functions

Binary compatibility

A library is **binary compatible**, if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile.

If a program needs to be recompiled to run with a new version of library but doesn't require any further modifications, the library is **source compatible**.

Binary compatibility saves a lot of trouble. It makes it much easier to distribute software for a certain platform. Without ensuring binary compatibility between releases, people will be forced to provide statically linked binaries.

Binary Compatibility Issues With C++
https://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++

Binary *in*compatibility

What breaks a library's ABI?

- Removing (or renaming) exported functions or classes
- Adding, removing or reordering member variables or virtual functions

There's an excellent list of Dos and Don'ts in the KDE Binary Compatibility Issues With C++ document

Designing APIs for stable ABIs

Consider defining module interfaces in terms of stable types

If your functions use simple types like `string_view` or `array_view` then they are insulated from the details of how `std::string` or `std::vector` (or your own sequence types) are defined

A more extreme form is to only define APIs in terms of `extern "C"` functions

Hack: dummy vtable entries

Library versioning

ELF shared libraries on Unix-like systems conventionally include a version number in the filename, e.g. `libfoo.so.3` and that filename is also the value of the `DT_SONAME` entry in the dynamic segment

The `libfoo.so.3` file may be a symlink to the actual library file with a more precise version numnber, such as `libfoo.so.3.0.4` or `libfoo.so.3.1.2`, but any libfoo.so with the same interface number of `3` should have the same SONAME of `libfoo.so.3` and be binary compatible, so can be used by any binary with a `DT_NEEDED` entry of `libfoo.so.3`

Library versioning

Some linkers allow versioning of individual symbols within a library

If `libfoo.so.3` exports a function `foo()`, the symbol could have a version appended to its mangled name, so code that links to the library has a reference to `_Z3foov@@LIBFOO_3.0` rather than `_Z3foov`

A later release of `libfoo` that changes the behaviour of `foo()` can define the function twice, so that `_Z3foov@@LIBFOO_3.0` continues to refer to a function with the old behaviour but `_Z3foov@@LIBFOO_3.1` is a separate symbol that refers to a function with the new behaviour

Inline namespaces

C++11 supports inline namespaces, allowing a type to be defined in an inner namespace but referred to in source code as if it were a member of the outer namespace

```
namespace lib {
  inline namespace v1 {
    class Widget;
  }
}
```

This gives the library author more control over the type, as it can be replaced in a later release by `lib::v2::Widget`, which has a different mangled name, but source code using the library just refers to `lib::Widget` and doesn't need to be changed

Inline namespaces

So inline namespaces allow a particular name in the source code to map to a different name for linkage purposes

But inline namespaces are not a complete solution

A class with a member variable of type `lib::Widget` changes its ABI if recompiled with `lib::v2::Widget` instead of `lib::v1::Widget`, the types of member variables are not represented in a type's mangled name

Libabigail

Compare two versions of a shared library, or compare a program with a new version of a shared library, to see if the new shared library has any ABI incompatibilities:

```
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 lib::func(const lib::Str&)' has some indirect sub-type changes:
    parameter 0 of type 'const lib::Str&' has sub-type changes:
      in referenced type 'const lib::Str':
        in unqualified underlying type 'struct lib::Str':
          1 data member insertion:
            'unsigned int lib::Str::capacity', at offset 96 (in bits)
```

`std::string` in GCC 5

```
namespace std {
#if _GLIBCXX_USE_CXX11_ABI
inline namespace __cxx11 __attribute__((__abi_tag__("cxx11"))) {
  template<typename C, typename Tr, typename A>
    class basic_string {
      // new SSO string ...
    };
}
#else
  template<typename C, typename Tr, typename A>
    class basic_string {
      // old COW string ...
    };
#endif
} // namespace std
```

The `abi_tag` attribute tells G++ that everything in the namespace relates to an ABI change, allowing it to warn about possible ABI problems related to use of those types, and to alter the name mangling of functions that return those types

The end

These slides are available at https://gitlab.com/wakelyaccu/abi

You can download and view the HTML version of these slides with commands like:

```
git clone https://gitlab.com/wakelyaccu/abi.git
firefox abi/abi.html
```