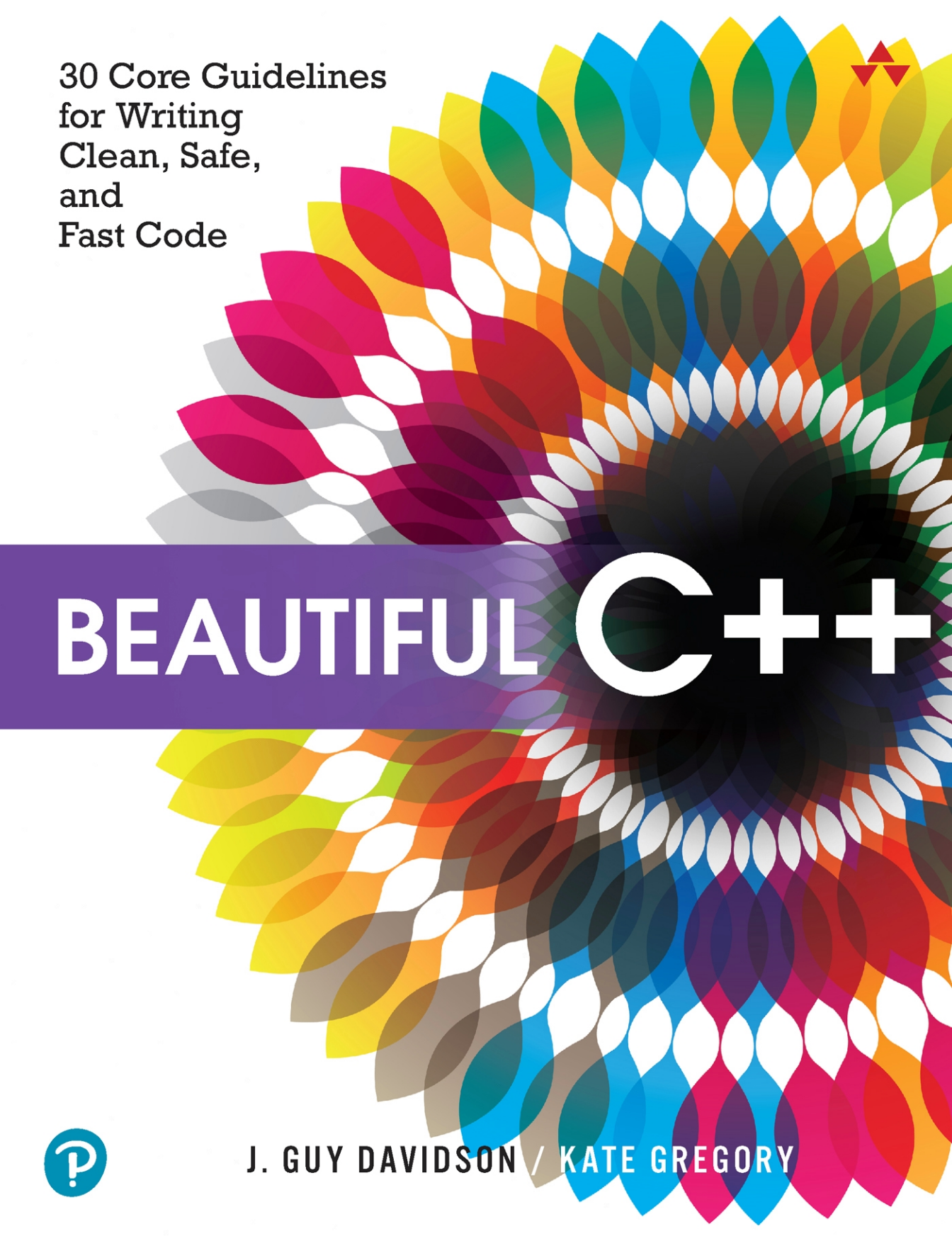
30 Core Guidelines
for Writing
Clean, Safe,
and
Fast Code
BEAUTIFUL C++
J. GUY DAVIDSON / KATE GREGORY

Beautiful C++

Chapter 2.3

I.26: If you want a cross-compiler ABI, use a C-style subset

Creating libraries

Writing libraries in C++ is a simple matter. Compile your source files, glue them together into a library, expose your exports in a header, or module if your compiler supports it, and supply the library file and header or module definition to your client.

Unfortunately, that is not the end of the story. There is a substantial amount of detail to get right. The purpose of this guideline is to show you a way to minimize this overhead, to prevent a truly overwhelming amount of work from landing on your desk at some random point in the future, and to enable you to create libraries that will serve the community for many years to come.

Recall the operation of the linker. It matches missing symbols from object files or libraries with exported symbols other object files or libraries. The declarations in the header point to definitions in the library file.

For example, imagine you have written a library that contains, among other items, the function

```
id retrieve_customer_id_from_name(std::string const& name);
```

and a header that declares it. This is a handy way of doing things: it allows you to create a package of functionality with a header, and your user does not need to spend time recompiling. Maybe your source is proprietary, and you are not allowed to share it.

The likely operation of this function is that it will retrieve the character buffer from the string reference, query a database Somewhere In The Cloud with it, and return an `id` formed from the result of the query. A string implementation may

contain a length followed by a pointer to a buffer, so retrieving the character buffer and passing it to the database is a trivial matter.

Now, imagine that your library becomes insanely popular: perhaps the database is full of particularly useful information and everyone wants a piece. You have hundreds, then thousands, then tens of thousands of clients, all cheerfully parting with their cash in exchange for this functionality. Suddenly, a trickle of complaints come in telling you that the library is crashing during the execution of this function. The trickle becomes a torrent and then a veritable flood of ire. You investigate and notice that there has been a toolset upgrade: the compiler, linker, and standard library have changed.

The next thing you do is rebuild the library and the unit tests with the new toolset and run everything. It all passes. There is no crash. You attempt to reproduce the bug report, but to no avail. What has happened?

The answer is that the definition of `std::string` has changed. When you first shipped the library, `std::string` was implemented as a length followed by a pointer. Unfortunately, your clients have upgraded to a new toolset which now implements `string` as a pointer followed by a length. They are now supplying a `std::string const&` to your function whose memory layout is different from what it is expecting. On attempting to dereference the pointer, it is in fact dereferencing the length, reaching into forbidden memory, and engendering a hasty exit.

Of course, when you recompiled the library and the unit tests, they were all built with the new toolset, so the definition of `std::string` agreed throughout, as a pointer followed by a length. Everything passed. It is only when new code called your old code that things went wrong.

What is an ABI?

What happened was that there was a change to the ABI. The API for your library did not change, but the ABI did. The term "ABI" may be new to you. It stands for application binary interface. Just as an API, an application programming interface, is a guide to humans about what you can do with a library, so is an ABI a guide to machines about how libraries interact. Think of it as the compiled version of an API.

An ABI defines several things: not just how objects are laid out in the standard library, but also how things are passed to functions. For example, the System V AMD64 ABI[1] specifies that the first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9. This ABI is followed by Unix and Unix-like operating systems. Other considerations include how exceptions are handled

1. https://wiki.osdev.org/System_V_ABI

and propagated, function prolog and epilog code, the `virtual` table layout, `virtual` function calling conventions, and so on.

A library that does not use types from another library in the signatures of its functions cannot suffer an ABI break if that library changes. Eventually, all libraries change, and this is a rather problematic fact of development.

The need for an ABI is one of the reasons why code compiled for one operating system will not work with another operating system: although they might have the same processor architecture, for example x86, they may have different ABIs. Of course, if there were one unifying ABI this would not matter, but ABIs are intimately bound up with the performance characteristics of the hardware. Pinning everything to one ABI would come at a performance cost. A library that does not use types from another library in the signatures of its functions cannot suffer an ABI break if that library changes. Eventually, all libraries change, and this is a rather problematic fact of development.

It is therefore important to keep your ABI stable. Changing a function in any way, the return type or the types, the quantity and order of arguments, or the `noexcept` specification, is an ABI break. It is also an API change. Changing definitions of data types or data structures in the private interface is not an API change but it IS an ABI break.

Even name mangling isn't immune from ABI breaks. An ABI can define a standard way of uniquely identifying the name of a function so that libraries built with different languages, for example C and Pascal, can be linked together. If you have ever seen the declaration `extern "C"` in a header file, that is a signal to the compiler that the function names within the declaration should have their names inserted into the exports table using the same naming scheme as a C compiler on that platform would use.

The earlier hypothetical problem with a library that takes a `std::string` is unlikely to trouble you. Today's linkers will simply prevent clients of different standard libraries from linking. There are a number of ways to do this: the ABI version can be embedded in the mangled name of the symbol, or the compiler can insert the ABI version into the object file and emit an error should anyone try to link conflicting versions together. The labor of dealing with conflicting ABIs is not eliminated, but it is exposed and added to the engineer's plate.

Recall the guideline: "If you want a cross-compiler ABI, use a C-style subset." We have now explained the nature of a cross-compiler ABI, and why observing it is important for both today and the future. Let us now move on to the C-style subset.

Paring back to the absolute minimum

So, what does a C-style subset consist of? The C types are often referred to as the built-in types. They are the types that are not made up of any other types. I would say they are the atomic types, but sadly that has another meaning in C++. You can, however, think of them as the fundamental building blocks of all your types. The list consists of[2]

```
void
bool
char
int
float
double
```

Some of these types can be modified by sign or by size. The keywords `signed` and `unsigned` can be applied to `char` and `int`. The keyword `short` can be applied to `int`. The keyword `long` can be applied to `int` and `double`. This keyword can also be applied twice to `int`. In fact, the keyword `int` can be omitted entirely and replaced with the sign or size modifiers. The modifiers can also be applied in any order: the type `long unsigned long` is entirely valid.

This might seem at first glance to be somewhat cumbersome, although if I saw `long unsigned long` in any code submitted for review, I would return it with a request for justification. The presence of the size modifier gives rise to a favorite interview question of mine: "How big is an `int`?" the answer to which is, "It depends." The standard offers some guarantees.

- A `short int` and an `int` are at least 16 bits wide.
- A `long int` is at least 32 bits wide.
- A `long long int` is at least 64 bits wide.
- 1 == `sizeof(char)` <= `sizeof(short)` <= `sizeof(int)` <= `sizeof(long)` <= `sizeof(long long)`.

The implementation decides on these widths. This set of choices is known as the data model. There are four models that have found wide acceptance:

- LP32 or 2/4/4 (`int` is 16-bit, `long` and pointer are 32-bit) as used in the Win16 API

2. https://en.cppreference.com/w/cpp/language/types

- ILP32 or 4/4/4 (`int`, `long`, and pointer are 32-bit) as used in the Win32 API and some Unix systems
- LLP64 or 4/4/8 (`int` and `long` are 32-bit, pointer is 64-bit) as used in the Win64 API
- LP64 or 4/8/8 (`int` is 32-bit, `long` and pointer are 64-bit) as also used in some Unix systems

This highlights the significance of the word "cross-compiler" in the guideline. Your library needs to work on your compiler today, but it also needs to work a year from now, and on the compiler used by your clients. If the compilers have different opinions on the size of an `int` or a pointer, then their behavior will differ in ways that you did not intend.

The floating-point types are simpler than the integral types, since they are defined by reference to another standard, particularly ISO/IEC/IEEE 60559:2011, which is the same as IEEE 754-2008. This makes them utterly reliable between platforms. `char` can be signed, unsigned, or neither; all three are distinct types. `unsigned char` is used for dealing with raw memory.

It is clear from this that changing data models is an ABI break: I first encountered ABI breaks (although I didn't understand the problem in this way) when I first installed Win32s on my Windows 3.11 machine in the early 1990s. Suddenly, all my `int`s took up twice as much memory and some of my programs fell over. More recently, the slight difference between LLP64 and LP64 (width of `long`) caused me pain writing for Windows and MacOS, the former using LLP64 and the latter using LP64.

You may now relax, breathe out, and raise an eyebrow. The advice is "If you want a cross-compiler ABI, use a C-style subset" and yet we have just demonstrated how fragile the built-in types are. However, this fragility is only present when you are trying to move between data models. Also, there is a way to overcome the data model hazard and that is with the use of fixed-width integer types. These embed the width of the type in the identifier, and have names like `int32_t` and `uint8_t`. These are defined in C in the header `<stdint.h>` and in C++ in the header `<cstdint>`.

To be clear, the original function declaration at the start of this chapter should not have taken a `std::string` across an ABI boundary, because the definition of string can change. It changed in GCC for C++11, and the pain was considerable. The function should have taken a `char*` and an `int`, or better still a `uint8_t`, because those will never change.

Exception propagation

This sounds like the job is done: pass and return fixed-width built-in types and there should be no problems. However, there is another way that data passes between functions, which is via the exception-handling machinery. When an exception is thrown, the compiler will allocate space for the exception and then start unwinding the stack via a call to a special function in the standard library implementation. On some platforms, this function will iterate through each catch statement via data inserted by the compiler corresponding to each function body. It's looking for a catch statement that will handle the type of exception that has been thrown.

There are two possible outcomes: either a matching catch statement is found, or no matching catch statement is found. In the latter case, the standard library implementation will call `std::terminate` which, by default, will call `std::abort`. This returns control to the host environment without cleaning up. If the stack unwinder is fortunate enough to find a matching catch statement, it will then go back through the stack and clean up each function, calling destructors and restoring the stack, before resuming execution at this matching catch statement. After the catch statement is executed, the memory allocated for the exception is released.

Setting aside the matter of what might happen when the exception is thrown because of a lack of memory, this behavior is deeply implementation specific. The mechanism used to allocate space for the exception, and the format of the data inserted by the compiler after each function body, will depend on the implementation, as will the means by which each catch statement is described to the stack-unwinding function. This information is all written by the toolset at compile time, and if two libraries built with two compilers differing by even the version number coexist in the same program, there is a risk of ABI incompatibility. The risk is quite small, however: the exception-handling info format is defined by the platform ABI, and the exception types are defined by the library, and the vendors go to great lengths not to change those things. However, if you are feeling particularly cautious, the only option here is to ensure that exceptions do not leave your library, and to decorate each function in the API with `noexcept` or `noexcept(true)` depending on your local style.

As you can see, some work is required to ensure a cross-compiler ABI for a library. This is a manifestation of the perils of binary dependency. You may decide to distribute (or consume) a library as source or as a prebuilt binary. In the example above the notion of proprietary code was observed, along with compilation time for your client. This can be a serious matter: a GUI library I use at my studio takes most of the day to configure and build. Binary versions of the library are available. I simply have to identify which version of the toolset I am using and press the download