BY DAVID CHISNALL

Clang in 10

FreeBSD 10 includes out-of-the-box support for the majority of the C11 and C++11 standards.

The road to get us to

this point has been long and has involved a lot of replacements of large parts of the toolchain. We're now shipping clang as the default C/C++ (and Objective-C) compiler and libc++ as the default C++ standard library implementation.

So why a new compiler and a new C++ standard library? The GNU Compiler Collection has been a part of FreeBSD for most of its life. There were painful upgrades from 2.x to 3.x and then to 4.x. It was always a somewhat sore spot for the project that a BSD-licensed operating system depended on a GPL'd compiler.

Some History

In 2007, GCC went to GPLv3. This license had one or two clauses that some major downstream consumers found unacceptable and so the decision was made not to import any GPLv3 code into the base system. The version of GCC stayed at 4.2.1.

In 2008, some developers at Apple released clang, a C front end for the LLVM compiler infrastructure. LLVM was originally offered to the Free Software Foundation as a new back end for GCC, but was turned down. Apple started using the LLVM back end (and hired the original developer of LLVM and a lot of other people) and continued to improve it.

LLVM is far more than just a C compiler. It provides a uniform intermediate representation that language front ends can generate, optimisers can modify, and back ends can convert into native code. One of its earliest was in Apple's OpenGL shader stack, where a naive LLVM-based JIT compiler outperformed the handwritten one by around 20% and worked

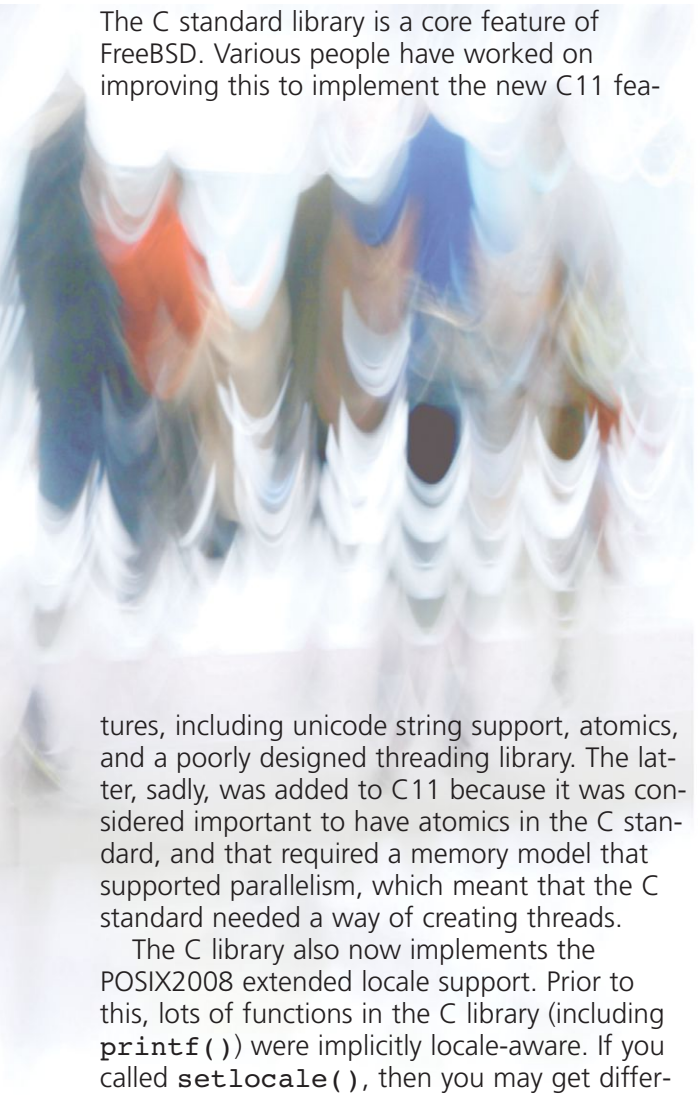
on all of Apple's supported architectures.

In the last five years, the combination of clang and LLVM has become a mature product. It's now the only compiler supported by Apple and is one of the standard compilers shipped with the Android NDK. Companies like ARM, Qualcomm, Apple, Google, AMD, Intel, and many others are contributing large amounts of code to it.

Meanwhile, our old GCC has begun to look quite dated. At the end of 2011, the C and C++ standards committees released specifications for new dialects of their respective languages. The extensions to C were relatively simple. The changes to C++ were huge. Both required changes to both the compiler and the standard library.

The C Standard Library

The C standard library is a core feature of FreeBSD. Various people have worked on improving this to implement the new C11 features, including unicode string support, atomics, and a poorly designed threading library. The latter, sadly, was added to C11 because it was considered important to have atomics in the C standard, and that required a memory model that supported parallelism, which meant that the C standard needed a way of creating threads.



The C library also now implements the POSIX2008 extended locale support. Prior to this, lots of functions in the C library (including `printf()`) were implicitly locale-aware. If you called `setlocale()`, then you may get different results from them. This includes fairly small things, like the separator character for floating point values. The `setlocale()` function sets the locale globally, which means that it's not safe for multiple locales in a multithreaded program.

The POSIX2008 locale extensions provide a per-thread locale, but more importantly they provide a set of variants of the standard C functions that take an explicit locale as a parameter. Lots of things use these APIs, including new versions of GNOME, but the primary consumer that we're interested in here is libc++.

Libc++ is another component of the stack to originate at Apple. Mostly developed by Howard Hinnant, it is a completely new implementation of the C++ standard library, designed from scratch for C++11. Implementing C++11 support in the standard library required quite invasive changes (which broke backwards compatibility) and so seemed like a good place to start from scratch. This also allowed all of the standard data structures to be redesigned in a way that makes more sense for modern hardware, for example focusing more on cache usage in std::string.

The Benefits of C++11 and C11

Now that we've got a new C++ stack and an improved C stack, what does that give us? I briefly mentioned some of the new features in C11, but my personal favorite is the new atomic types. These are declared with the `_Atomic()` type qualifier, for example:

```
_Atomic(int) x;
```

The qualifier ensures that you don't accidentally mix atomic and non-atomic accesses to the same variable. Simple operations will automatically become atomic. Some care is needed here. For example, consider the following three lines:

```
x += 1;
x++;
x = x + 1;
```

The first two will be a sequentially-consistent atomic increment. The last, however, will be a sequentially-consistent atomic read, an add, and then an atomic store. Sequentially consistent means that atomic operations are all globally visible in the same order. For example, if one core increments x and another increments y, then either the increment to x or y is visible first, but it is the same on all cores.

At the opposite extreme, atomic operations with relaxed ordering require that the operation be atomic with respect to that single variable, but not with respect to others. If x and y were relaxed, then it would be acceptable for some threads to see the new x and the old y, and some to see the new y and old x.

The stdatomic.h header contains a lot of functions that operate on atomic variables, and our implementation contains several code paths to make it work with old compilers. This is a pattern that we've replicated elsewhere and things like the `_Thread_local` storage qualifier and similar are implemented in our standard headers using extensions when using a compiler that does not support them natively.

One other addition in C11 has made it possible to clean up some of our headers. The standard adds `_Generic()` expressions, which are similar to switch statements selecting based on the type, rather than the value, of an expression. This is only useful in macros, but it's useful in several standard macros that must be defined in C header files. In particular, there are several related to numerics that are defined for all floating-point types.

Two examples in this category are `isinf()` and `isnan()`, which return true if the argument is infinite or a not-a-number value, respectively. Our old code was determining the correct path to call depending on the size of the argument. This meant that if you passed a 32-bit `int` value, it would call the function that expected a 32-bit `float`. This would always return false (because no `float` that is created by casting an int can possibly be infinite or not-a-number), but almost certainly hid a logic bug because there's no reason why you'd ever want to check these properties on an int.

We now use `_Generic()` for these and so they will always go to the correct function and you get an error if you try to call it with the wrong argument. We found a few bug ports as a result, and some quite bizarre behavior. For example, both Google's v8 and Mono had configure script checks that tested whether `isnan(1)` worked. In the Mono case, if they detected that `isnan(int)` didn't work, then they declared their own `isnan(double)` to use, which then conflicted with the system one.

Challenges with Clang

Getting the system ready for clang, initially as the system compiler and then as the only compiler in the base system, has been an interesting experience. A lot of the initial experiences were simply from getting FreeBSD to compile without warnings. Clang gives a lot more warnings than our old gcc (gcc 4.8 is now at a similar quality, although still has a slight tendency toward false positives) and we try to ensure that all of our code builds without warnings. Somewhat amusingly, the worst offender in our tree for having compiler warnings was gcc itself.

In the ports tree, things were somewhat different. We do maintain local patches for a lot of programs, but ideally these should be small (and should be pushed upstream where possible).

Most C code works fine with clang. The biggest issue that we faced in the ports tree was that clang defaults to C99 as the dialect when invoked as `clang` or `cc`, whereas gcc defaults to C89. It's somewhat depressing that people still invoke a C compiler as cc in 2013, because cc was deprecated in the 1997 release of POSIX and defined as accepting an unspecified version of the C language. Back then, the choices were K&R C or C89. If you wanted C89, you were recommended to invoke the `c89` utility. The next release of POSIX added a `c99` utility. Presumably the next one to be published will also specify a `c11` utility.

The C99 specification was carefully designed so that valid C89 programs were also valid C99 programs, so this shouldn't have been a problem. Unfortunately, this didn't quite work because few people wrote C89 code, instead they wrote C89 with GNU extensions. I said GCC defaulted to C89 mode, but that's not quite correct: it defaulted to C89 with GNU extensions (`gnu89`, as it's known on the command line).

There is only one significant incompatibility between C89 with GNU extensions and C99, and that's the handling of the `inline` keyword. The differences meant that code that expected the GNU inline rules would end up with functions defined multiple times in C99 mode and so would fail to link. This is relatively easy to fix—just add `-fgnu89-inline` to the compiler flags—but it needed to be done for every port that had this kind of error. When you have over 20,000 ports, even simple fixes are a lot of work.

In C++, the problems were more pronounced. The rules for symbol resolution in C++ are incredibly complicated. This is especially true inside templates, where the standard calls for a two-stage lookup. Both GCC and the Microsoft C++ compiler managed to get this wrong. Of course, they did it in different wrong ways,

which was why it has traditionally been very difficult to move C++ code between compilers.

Clang benefited from all of this experience and wrote the C++ parser to the letter of the specification. This means that any standards-compliant C++98 code will compile with clang. These days, so does C++11 code, and some C++1y code (C++1y is the draft that will most likely become C++14). Unfortunately, when you refuse to compile a popular open source program, you don't get much sympathy when you turn around and say—"well, the code is invalid."

The Challenge of Migration

For C code, there is no difficulty migrating. The C ABI is defined by the target platform and both Clang and GCC generate entirely compatible code. This is also true for C++ code, if you're only talking about C++—the language. Unfortunately, there is more to either language—there is also the standard library. In the case of C, this is FreeBSD libc. Again, this is shared between compilers.

In the case of C++, it's actually two libraries. The smaller of the two implements the dynamic parts of the language such as exceptions and the `dynamic_cast<>` operator. The larger implements the standard template library (STL). In our old stack, these were implemented by libsupc++ and libstdc++, respectively. Originally, these two were statically linked together.

In the new stack, these are libcxxrt and libc++. As part of the migration path, we wanted to make it possible for programs to link against libraries that used both libc++ and libstdc++. This required modifying our libstdc++ to link against libsup++ as a dynamic (shared) library, which then allowed libmap.conf to switch between them.

Unfortunately, life is never that simple. ELF symbol versioning associates the symbol with both the version and the library that it came from and so existing binaries would fail to link on symbols suddenly moved from `libstdc++.so` to `libsupc++.so`. The solution to this was to make `libstdc++` a filter library. This allows it to, effectively, forward the symbol resolution on to libraries it linked against.

With this done, it became possible to link against both. The STL symbols have different names and so you will use the ones from whichever headers you included in the source code. Unfortunately, because they have different symbol names (and different binary layouts), you can't use them interchangeably. If you have a library that has interfaces that use STL types (for example, `std::string`) then both the library and the things that call it must use the same STL implementation.

This causes some problems in the ports collection, because a few libraries won't compile with clang and so can't use libc++, whereas others require C++11 and so won't compile with our base GCC. Using a GCC from ports doesn't really address this either, as many of the old C++ programs also won't compile with a new GCC, and the new libstdc++ is not binary-compatible with the one that we have in the base system either.

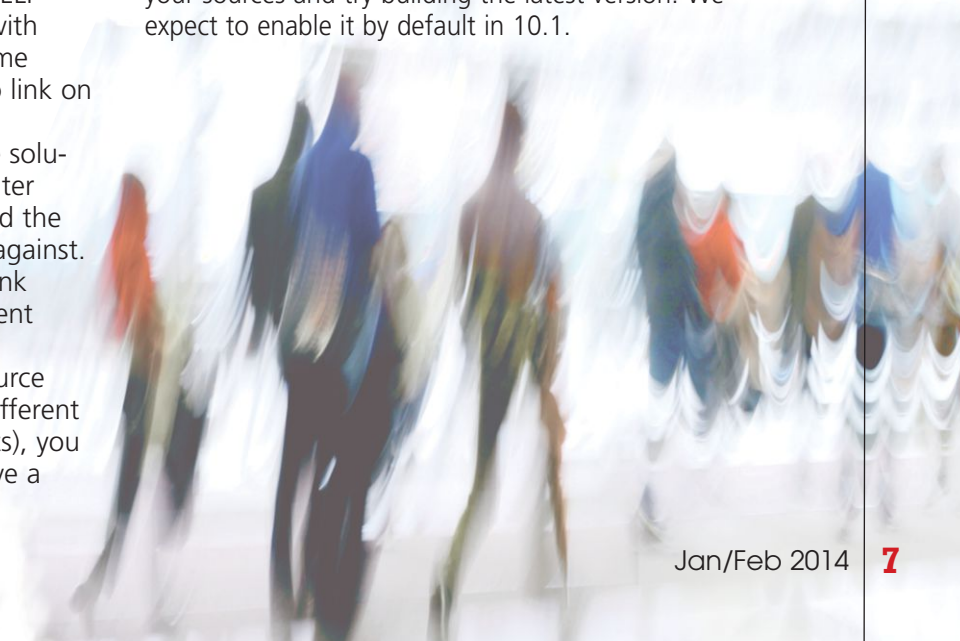
Debugging

One other unfortunate problem with the clang switch is that clang now emits debug information conforming to version 4 of the DWARF standard. Soon, it will default to DWARF 5, which includes support for much smaller debug info tables and for separating out the debug info into separate files during compilation so that they can be linked separately.

Unfortunately, the old version of the GNU debugger (GDB) that we include in the base system can only support DWARF 2. For 10, we've imported the LLVM Debugger (LLDB) and Ed Maste has been working (with FreeBSD Foundation funding) on the FreeBSD port.

LLDB, like the rest of LLVM, is very modular. It is intended as a set of libraries that allow debugging features to be added to various applications, ranging from command-line tools to IDEs. It is largely developed by Apple and so remote debugging was a core part of the design, allowing ARM devices to be debugged from x86 desktops.

All of these are nice features, but unfortunately LLDB isn't quite ready for enabling by default in the 10 release. It's in the tree, so feel free to upgrade your sources and try building the latest version. We expect to enable it by default in 10.1.



Clang in 10

Architectural Problems

These days, FreeBSD has one tier 1 architecture: x86 (in 32-bit and 64-bit variants). ARMv6 and newer are very close to being tier 1 as well. These two are well supported by Clang and by the LLVM back end. Unfortunately, we also have a lot of tier 2 architectures, such as SPARC, PowerPC, and MIPS, with less good support.

LLVM has quite good support for 64-bit PowerPC, developed largely by Argone National Laboratory, but not nearly as good support for 32-bit PowerPC. Since Apple switched to Intel, these architectures have been dead in the consumer PC market, but they're still popular in a lot of places where people ship embedded FreeBSD-derived systems, especially in the automotive industry.

The MIPS back end is now able to compile LLVM itself, which is quite an achievement given the size and complexity of the code base, but it's still a little way away from being able to compile the FreeBSD kernel.

SPARC and IA64 are two with less certain futures. The SPARC back end in LLVM is one of the oldest ones, yet it is still not production-ready. The Itanium back end was removed, after being unmaintained for a while. Intel doesn't seem to be pushing Itanium very hard, and Oracle seems to regard SPARC as a platform for running Solaris-based Oracle appliances, so the future of these architectures is not that certain anyway, but it would be a shame to drop support for them in FreeBSD while there are still systems using them.

Going Forward

The goal in all of this was to make FreeBSD a modern development platform. We've achieved that. FreeBSD 10 shipped with the most complete C11 and C++11 (and C++1y) implementations of any system to date. We now have a modern compiler and C++ stack, with an active upstream community that is engaged with FreeBSD as a consumer, and a number of people (myself included) who contribute to both projects.

We still have a few missing pieces for a completely BSD-licensed toolchain, however. We currently ship a lot of GNU binutils. Some things, such as the GNU assembler, are easy to replace. The LLVM libraries contain all of the required functionality; they just require small tools to be written to implement them.

The one exception is the linker. Like compilers, linkers are quite complex pieces of software. We're currently evaluating two linkers to replace GNU ld. The first, MCLinker, was originally developed by MediaTek using LLVM libraries and now has a larger community. It currently ships, was one of the linkers in the Android SDK, and can link all of the base system, but lacks support for symbol versioning (this may have been finished by the time you're reading this, as work is ongoing to implement it).

The other option is lld, the LLVM linker. This is a more complex design and is not yet as advanced, but does have some large corporate backers such as Sony (Sony is a FreeBSD consumer), and so might be a better long-term prospect.

Whichever we select, FreeBSD will continue to pick the best tools for the job. We hope to have a fully BSD licensed toolchain by default for 11.0, and as optional components in the 10.x series. Being BSD licensed is always nice, but we won't switch until the tools are also better. ●

David Chisnall is a researcher at the University of Cambridge Computer Laboratory and a member of the FreeBSD Core Team. He is also an active contributor to several other open source projects, including LLVM, GNUstep, and Étoilé. In between writing code, he has written several books. When not in front of a computer, he dances Cuban salsa and Argentine tango.