DDMD and Automated Conversion from C++ to D

Daniel Murphy (aka ‘yebblies’)

About me

- Using D since 2009
- Compiler contributor since 2011

Overview

- Why convert the frontend to D
- What's so hard about it
- What happened to previous attempts
- How magicport works
- Future of (D)DMD

WHY CONVERT THE FRONTEND TO D?

- “The point is not to use the compiler to stress test the language. NOT AT ALL. The point is to improve the compiler by taking advantage of what D offers.” – Walter Bright

- D is much nicer to work with than C++
- Refactoring is easier
- Avoid wasting time on C++ limitations
- Take advantage of powerful features to improve performance

The Challenge

- Frontend is pretty big
  - Currently ~120k lines
- Rapidly changing
  - ~20 pull requests per week
- Inevitable problems make estimating time difficult
  - Pausing development for months is undesirable

Past approaches

- Port by hand (original DDMD)
- Rewrite from scratch (SDC)

Hand Port

- Compiler is big
- More work added every day as pull requests are merged
- Uncontrollable urge to refactor/rearrange
- High probability of introducing bugs

- Theoretically possible, never successfully finished

Re-write from scratch

- Chance to do a clean, new design!
- Iron out errors in the spec

- Lose work done on implementing complex features (but keep the test suite)
- Compiler is big
- Huge amount of work compared to direct porting

- SDC is being actively developed
- Completion time is uncertain

A NEW APPROACH

- Automatically convert source
- Development continues non-stop on original
- Switch to D version only when generated code is good enough

Automatic conversion – Attempt 1

- Tokenize source (after pre-processing)
- Search and replace patterns
  - `id '->' id` becomes `id '.' id`
- Simple to implement
- Gets 95% of the way there

AUTOMATIC CONVERSION – ATTEMPT 1

- Source after pre-processing means only one platform can be supported
- Last 5% is made entirely of special cases
- Even basic semantic analysis is very difficult
- Had to resort to hardcoding variable names for some rules

- Too hard
- Gave up
- Could be used to assist hand porting

Automated conversion – Attempt 2

- Parse C++ source
- Adjust AST
- Write out D source

- C++ is really hard to parse
- Really, really hard
- Pre-processor is not part of C++ (but we have to parse it anyway)

Aim lower

- Don’t accept all C++ code
- Don’t have to handle invalid code
- Build list of types before parsing
  - What is `a * b;` ?
  - Depends on what symbols a and b are
- Some tricky cases can be special-cased
- Don’t support templates (except Array)

Making Things Easier

- We can cheat!
- Style can be normalized in C++ source
- Can change the source to use features that are easier to convert
- Manually port tricky parts instead of supporting more features
  - Array
  - SignExtendedNumber

CONVENTIONAL WISDOM

- “My experience chiming in - never ever ever attempt to refactor while translating. What always happens is you wind up with a mess that just doesn't work.” – Walter Bright
- Rules are different for automatic conversion
- Translating takes < 10 seconds
- If it doesn’t work, throw away the result and try again

Outcome

- Lots of changes made to C++ source
- Automatic porting then worked on 97%
- 10 files manually ported
  - Templates
  - Operator overloading
  - OS/Memory/low level code

Magicport

- C++ to D source to source compiler
- Some very basic analysis of code
- D pretty printer
- ~6000 lines (of horrific hacks)

Limitations

- Tool is single-use
- Makes lots of assumptions about code
  - No variables have the same names as types
  - Multi-var declarations will have a single type
- Many translations hard-coded
  - #defined values become manifest constants
  - Macros are re-written as template functions

Lexing

- Tokenize source
  - Very simple
  - Assume ASCII
  - Doesn’t need to be efficient
  - Recognize pre-processor constructs as tokens (e.g. ‘#ifdef’)

TYPE LIST

- Scan through tokens looking for type names
  - Match patterns
    - ‘class’ ? ‘;’
    - ‘struct’ ? ‘{‘
    - ‘typedef’ ‘Array’ ‘<‘ ‘class’ ? ‘*’ ‘>’ ? ‘;’
  - Build list to make parsing easier

PARSING

- Parse our version of C++
  - 25% of total code
  - Limited subset of C++
    - E.g. Can't handle function pointer types in many places
  - No error recovery
  - Builds simplified AST
    - `a.b / a->b / a::b` all produce `a.b`

Analysis

- Build lists of class declarations, call expressions, etc.
- Check that all types in list are referenced
- Count declarations inside #ifdef blocks
- Remove duplicate declarations (typedef)

MERGING

- Merge function declarations
  - Take body from definitions
  - Take default arguments from forward declarations
  - Check for mismatches or duplicates
- Same thing for static member variables

SPECIAL CASE

- Scope has a default constructor
- Automatically convert it to default member initializers

Strip Out Dead Declarations

- #includes
- Empty version blocks
- #undef
- Include guards
- Default ctors
- ‘extern’ function prototypes

Collect Declarations

- Build hash map containing all top-level declarations
- Use simple mangling scheme
  - ‘function importHint’
  - ‘struct Loc’
  - ‘enum LINK’
- Include parameter names for overloaded functions

D GENERATION

- List of modules and members in json file
  - List of imports
  - List of members (using mangled name)
  - Extra D code (e.g. "extern (C++) Library LibElf_factory();",)
- Write out each file
- Error on unknown declarations
- Error on unreferenced declarations

#IFDEF ISSUES

- #if doesn't follow language grammar

```
if (x
#if SOMETHING
      && y
#endif
       )
```

- Difficult to parse
- Sometimes impossible

#ifdef Issues

- Cheat!
- Just change the C++ source to something valid in D

```
if (x && (!SOMETHING || y))
```

- Usually very straightfoward
- C++ code generally benefits from this too

Comment Issues

- Can (and do) appear anywhere

```
if (x && y /*&& z*/) { }
if (x)
    /*doSomething()*/;
```

- Lots and lots of special cases to parse correctly
- Instead, parse the most common cases
- Remove rest from C++ source

Converts successfully!

- Generated code doesn’t compile
- Local variable shadowing is illegal in D
- Implicit narrowing conversion is an error
- Class handles don’t convert to void pointers
- No implicit struct construction

e.g.

```
    void func(Loc loc);
    func(0);
```

CAN'T COMPILE

- D string literals are passed to varargs functions as arrays
- D checks for goto skipping variable initializations are much stricter that C++
- sizeof(arr)/sizeof(arr[0]) doesn't work in D
- #defines are not scoped
- String literals are type-checked
  - `char *s = “Don’t ever do this”;`
- All ‘fixed’ in C++ source

D'S LIMITATIONS

- No struct default constructors
  - Re-wrote structs so default initializers were all zero (except Scope)
- version() is much less powerful than #if
  - `version(A || (B && C))`
  - Used static if instead
- No way to define data from command line
  - Like `-DNAME=VALUE`

It compiles!

- But AST classes will need to be accessed from C++ glue layer

- Added missing support for C++ classes
- Allowed non-virtual C++ member functions
- Allowed C++ member variables

- Now we can try linking against the glue layer

C++ Mangling Issues

- Linker error everywhere
- struct and class have different mangling
- C++ has three char types – which one to use?
  - Defined our own utf8_t
- uint64_t is not always the same type
  - unsigned long – freebsd64, linxu64, osx64
  - unsigned long long - *32, win64
- size_t is not always the same type
  - unsigned int – win32, linux32, freebsd32
  - unsigned long – osx32
- Solution – drop osx32 dmd binary support

C++ ABI Issues

- It then links, but crashes

- Member layout/alignment mismatches
  - Generate code to check offsets
- Calling convention mismatches
  - Fuzz tester
- vtbl layout (win32)
  - Overloaded functions are reversed in vtbl
- Varargs problems
  - Argument passing wrong on posix64 and win64
  - va_copy doesn't work on posix64

OTHER BACKEND BUGS

- ~8 codegen bugs found in DMD backend
- DMD is not idiomatic D
  - Exercises a ‘new’ subset
- Tough to reduce and tough to fix

OUTSTANDING ISSUES

- FP returns broken on win32 (DMC and/or DMD)
- Still have struct passing bugs on posix64
- Constructor and destructor calls do not work across language boundary

- All worked around!

Cosmetic Issues

- D doesn't support out-of-class function definitions
  - Move compiler passes to visitor interface
  - Allows keeping layout the same in C++ and D
  - Allows backends to add passes without modifying frontend classes (sometimes)
- Minor array/string/comment formatting issues
  - dfmt might be able to fix these one day
  - Could just fix them after transition to D

WHERE NEXT?

- Fix remaining performance issues
  - ~20% hit due to compiling with DMD vs GCC
- Clean up generated code
- Wait for GDC/LDC to catch up to 2.067
- Delete C++ code and switch

- Port DMD glue layer to D
- Get GC working with DDMD
  - Requires all allocations be done through GC
- Remove backend-dependent code from frontend

PULL REQUESTS WILL BREAK

- Most can be automatically updated
  - Rebase on top of last C++ commit
  - Automatically convert to D
  - Diff against first D commit
  - Rebase on top of latest master
- Not significantly harder than rebasing to fix a normal conflict

Timeline

- Started experimenting – 2012
- Forum thread: ‘Migrating dmd to D?’ – February 2013
- First commit – March 2013
- Zero link errors – June 2013
- All ‘compilable’ tests pass – July 2013
- Self-hosts on win32 – July 2013
- Self-hosts on linux – December 2013
- Can use unpatched master as host and source – February 2014

Timeline

- Linux DDMD goes green on autotester – July 2014
- All platforms green on autotester – February 2015
- Magicport and manually ported source merged into master – April 2015

- > 2 years
- 398 pull requests – over 8% of total dmd pull request

MAGICPORTING OTHER PROJECTS

- Must use a small, consistent subset of C++
- Need easy access to refactor the C++ source
- Doesn't rely too heavily on the preprocessor

- Must be comfortable debugging memory corruption
  - This will get better in the future
- Must have good understanding of low-level C++ details

- Well worth the effort!

QUESTIONS?