GDC:
The GNU D Compiler

Iain Bucław

@ibuclaw

GHM 2013
1. History of Porting D Front End (DFE)

2. GDC Current Status

3. The Anatomy of a GCC Front End

4. GDC Extensions

5. Future Plans
What is D?

- D is a language with C-like syntax and static typing.

- Pragmatically combines convenience, power, and efficiency.
  - High efficiency
  - Systems level access
  - Modeling power
  - Simplicity
  - High productivity
  - Code correctness
Features Inherited from C/C++

- General look and feel of C/C++.
- Object Oriented.
- Template Metaprogramming.
- Exception Handling.
- Runtime Type Identification.
- Operator Overloading.
Features Dropped from C/C++

- C source compatibility.
- C Preprocessor and Macros.
- Multiple Inheritance.
- Forward Declarations.
- Support for 16-bit architectures.
- Implementation Specific Types.
A Short History of Porting the D Front End.
History

Late/1999:
   Work began on D.

August/2001:
   First public announcement and draft specification.

December/2001:
   D v0.01 Alpha released.
History

January/2002:

Early discussions of wanting to port D to GNU/Linux began.

April/2002:

Walter Bright releases D Front End sources.

May/2002:

Birth of D.gnu Mailing List and BrightD Compiler Project.

June/2002:

OpenD Compiler Project announced.
August/2002:
   D Linux (DLI) released.

May/2003:
   Walter Ports DMD to GNU/Linux.

February/2004:
   GDMD Compiler Released.

March/2004:
   DGCC Compiler Released.
History

September/2007:
New Development of an LLVM D Compiler.

June/2008:
DGCC Development Abandoned.

September/2009:
GDC Revival Project Kicks Off.

December/2009:
Enter Your Humble Speaker.
Current State of D2 Compiler

- Three main compilers based off the D2 Front End.
- Platform support for Linux, FreeBSD, OSX, Solaris, and Windows.
- Target support for ARM, PowerPC, x86, x86_64.
- D Runtime gaining support for more targets.
- Phobos becoming platform agnostic.
Current GDC Support Status.
GDC: Language Support

- D Front End 2.063.2.

- Passes 100% on D2 Testsuite on x86 and x86_64.

- Passes all unittests in Druntime and Phobos.
GDC: Target Support

- **x86/x86_64**: Solid support.
- **ARM**: Partial support.
- **MIPS**: Partial support.
- **Others**: Untested / No runtime support.
GDC: Platform Support

- **GNU/Linux**: Main support platform.
- **FreeBSD/OpenBSD**: Support should be there.
- **OSX**: Lacks TLS Support.
- **Windows/MinGW**: Alpha quality release available.
GDC: Incompatibilities with DMD.

- GDC follows the D calling convention as per the spec.

- Except for Win32, which defines the D calling convention.

- No D Inline Assembly implemented.

- No naked function support.
GDC: Incompatibilities with DMD.

- Type va_list matches C ABI.

- No __simd support.
  - Allow __vector sizes of 8, 16 or 32 bytes.
  - No current restrictions on what targets can use __vector.

- **gcov** and **gprof** replace -cov and -profile.

- **gdmd** script maintained separately.
The Anatomy of a GCC Front End.
Why GCC?

- GCC is developed to be 100% free software.

- The entry barrier to GCC development has gotten considerably lower during the last few years.

- With work on documentation and separation of internal modules, writing your own front end for GCC has become accessible to a wider community of developers.
Able to translate from a variety of source languages to assembly.

Encapsulated into one command.

Front end is made up of two main components.
Compilation Driver

- User interfacing application.
- Knows about all supported languages.
- Able to determine source language.
- Passes output between compiler and assembler.
void
lang_specific_driver (struct cl_decoded_option **in_decoded_options,
                      unsigned int *in_decoded_options_count,
                      int *in_added_libraries)
{
}

int
lang_specific_pre_link (void)
{
    return 0;
}

int lang_specific_extra_outfiles = 0;

const struct spec_function lang_specific_spec_functions[] =
{
    { 0, 0 }
};
One compiler proper for each language.

Composed from three components.
The Front End contains all the language processing logic.

The Middle End is the platform independent part of the compiler.

The Back End is then the platform dependent part.
- GENERIC is a tree language.

- Mechanism to define own node types.

- Supports everything there is to represent in a typical C function.

- During the course of compilation, it is lowered into an intermediate code called GIMPLE.
GIMPLE is a subset of GENERIC.

Breaks down all expressions, using temporaries to store intermediate results.

Further transforms all blocks into gotos and labels.

Lowered down to RTL, or Register Transfer Language.
Interfacing with D Front-End

- GDC initialises the D Front-End, sets up all global parameters.
- D Front-End parses and runs semantic on the code.

```c
void Import::semantic(Scope *sc);
void Module::semantic();
void Declaration::semantic(Scope *sc);
void Dsymbol::semantic(Scope *sc);
Type *Type::semantic(Loc loc, Scope *sc);
Expression *Expression::semantic(Scope *sc);
Statement *Statement::semantic(Scope *sc);
Initializer *Initializer::semantic(Scope *sc, Type *t);
```
GDC generates GENERIC to be sent to backend.

```c
void Module::genmoduleinfo (void);
void Declaration::toDt (dt_t **pdt);
void Declaration::toObjFile (int);
void Declaration::toSymbol (void);
void Dsymbol::toObjFile (int);
void Dsymbol::toSymbol (void);
type* Type::toCtype (void);
dt_t** Type::toDt (dt_t **pdt);
elem* Expression::toElem (IRState *irs);
void Statement::toIR (IRState *irs);
dt_t* Initializer::toDt (void);
```
GCC backend compiles down to RTL.

```c
static void
d_write_global_declarations (void)
{
  tree *vec = (tree *) globalDeclarations.data;
  // Complete all generated thunks.
  cgraph_process_same_body_aliases ();
  // Process all file scopes in this compilation, and the external_scope,
  // through wrapup_global_declarations.
  wrapup_global_declarations (vec, globalDeclarations.dim);
  // We’re done parsing; proceed to optimize and emit assembly.
  if (!global.errors && !errorcount)
    finalize_compilation_unit ();
  // Now, issue warnings about static, but not defined, functions.
  check_global_declarations (vec, globalDeclarations.dim);
  // After cgraph has had a chance to emit everything that’s going to
  // be emitted, output debug information for globals.
  emit_debug_global_declarations (vec, globalDeclarations.dim);
}
```
Interfacing with GCC

```
#define LANG_HOOKS_NAME         "GNU D"
#define LANG_HOOKS_INIT         d_init
#define LANG_HOOKS_INIT_OPTIONS d_init_options
#define LANG_HOOKS_OPTION_LANG_MASK d_option_lang_mask
#define LANG_HOOKS_HANDLE_OPTION d_handle_option
#define LANG_HOOKS_POST_OPTIONS d_post_options
#define LANG_HOOKS_PARSE_FILE    d_parse_file
#define LANG_HOOKSTYPES_COMPATIBLE_P d_types_compatible_p
#define LANG_HOOKS_BUILTIN_FUNCTION d_builtin_function
#define LANG_HOOKS_BUILTIN_FUNCTION_EXT_SCOPE d_builtin_function
#define LANG_HOOKS_REGISTER_BUILTIN_TYPE d_register_builtin_type
#define LANG_HOOKS_FINISH_INCOMPLETE_DECL d_finish_incomplete_decl
#define LANG_HOOKS_GIMPLIFY_EXPR d_gimplify_expr
#define LANG_HOOKS_EH_PERSONALITY d_eh_personality
#define LANG_HOOKS_EH_RUNTIME_TYPE d_build_eh_type_type
#define LANG_HOOKS_WRITE_GLOBALS d_write_global_declarations
#define LANG_HOOKS_TYPE_FOR_MODE d_type_for_mode
#define LANG_HOOKS_TYPE_FOR_SIZE d_type_for_size
#define LANG_HOOKS_TYPE_PROMOTES_TO d_type_promotes_to

struct lang_hooks lang_hooks = LANG_HOOKS_INITIALIZER;
```
enum built_in_attribute
{
#include "builtin-attrs.def"
    ATTR_LAST
};

static tree built_in_attributes[(int) ATTR_LAST];

static void
d_init_attributes (void)
{
#include "builtin-attrs.def"
}
enum d_builtin_type
{
    #include "builtin-types.def"
    BT_LAST
};

static tree builtin_types[(int) BT_LAST + 1];

void d_init_builtins (void)
{
    #include "builtin-types.def"

    d_init_attributes ();

    #include "builtins.def"

    targetm.init_builtins ();
    build_common_builtin_nodes ();
}
void d_backend_init (void)
{
    init_global_binding_level ();

    // Parameters are (signed_char = false, short_double = false).
    build_common_tree_nodes (false, false);
    d_init_builtins ();

    if (flag_exceptions)
        d_init_exceptions ();

    main_identifier_node = get_identifier ("main");
}
module demo;

int add(int a, int b)
{
    return a + b;
}
Module (demo);

FuncDeclaration (demo.add)
CompoundStatement {
    ReturnStatement { AddExp (SymbolExp (a) + SymbolExp (b)); }
}
```c
.demo.add (int a, int b) 
{
    return <retval> = a + b;
}
```

```c
.demo.add (int a, int b) 
bind_expr (  
    return_expr (  
        init_expr (<retval>, plus_expr (a, b))  
    )
)
```
demo.add (int a, int b)
{
    int vartmp0;
    vartmp0 = a + b;
    return vartmp0;
}

demo.add (int a, int b)
gimple_bind (    int vartmp0;
    gimple_assign (plus_expr, vartmp0, a, b)
    gimple_return (vartmp0)  )
module demo;

long fib (uint m) {
    return (m < 2) ? m : fib (m - 1) + fib (m - 2);
}
Module (demo);

FuncDeclaration (demo.fib)
CompoundStatement {
    ReturnStatement {
        CondExp { CmpExp (SymbolExp (m) < IntegerExp (2))
            ? SymbolExp (m)
            : AddExp (CallExp (fib, (MinExp (SymbolExp (m),
                        IntegerExp (1)))),
                       CallExp (fib, (MinExp (SymbolExp (m),
                        IntegerExp (2))))); }
    }
}
demo.fib(uint m)
{
    return <retval> = m <= 1 ? (long) m : demo.fib (m - 1) + demo.fib (m - 2);
}

demo.fib(uint m)
bind_expr (return_expr (init_expr (<retval>,
    cond_expr (le_expr, m, 1,
        nop_expr (m),
        plus_expr (call_expr (demo.fib, minus_expr (m, 1)),
            call_expr (demo.fib, minus_expr (m, 2)))
    )
    )
)
demo.fib (uint m)
{
    long vartmp0;
    long iftmp0;
    uint vartmp1;
    long vartmp2;
    uint vartmp3;
    long vartmp4;
    if (m <= 1) goto L1; else goto L2;
L1:
    iftmp0 = (long) m;
    goto L3;
L2:
    vartmp1 = m + 4294967295;
    vartmp2 = demo.fib (vartmp1);
    vartmp3 = m + 4294967294;
    vartmp4 = demo.fib (vartmp3);
    iftmp0 = vartmp2 + vartmp4;
L3:
    vartmp0 = iftmp0;
    return vartmp0;
}
demo.fib (uint m)
gimple_bind (
   long vartmp0;
   uint vartmp1;
   long vartmp2;
   uint vartmp3;
   long vartmp4;
   long iftmp0;
   gimple_cond (le_expr, m, 1, (L1), (L2))
   gimple_label (L1)
   gimple_assign (nop_expr, iftmp0, m)
   gimple_goto (L3)
   gimple_label (L2)
   gimple_assign (plus_expr, vartmp1, m, 4294967295)
   gimple_call (demo.fib, vartmp2, vartmp1)
   gimple_assign (plus_expr, vartmp3, m, 4294967294)
   gimple_call (demo.fib, vartmp4, vartmp3)
   gimple_assign (plus_expr, iftmp0, vartmp2, vartmp4)
   gimple_label (L3)
   gimple_assign (var_decl, vartmp0, iftmp0)
   gimple_return (vartmp0)
)
GDC Extensions
Custom Static Chains

- Generated for all nested functions
- Generated for toplevel functions with nested references.

```c
int delegate() foo()
{
    int x = 7;

    int bar()
    {
        int baz()
        {
            return x + 3;
        }
        return baz();
    }
    return &bar;
}
```
closure.foo.bar.baz (void *this)
{
    return <retval> = ((CLOSURE.closure.foo *) this)->x + 3;
}

closure.foo.bar (void *this)
{
    return <retval> = closure.foo.bar.baz ((CLOSURE.closure.foo *) this);
}

closure.foo (void *this)
{
    int x [value-expr: (__closptr)->x];
    struct CLOSURE.closure.foo *__closptr;

    __closptr = (CLOSURE.closure.foo *) _d_allocmemory (8);
    __closptr->__chain = 0B;
    __closptr->x = 7;
    return <retval> = {.object=__closptr, .func=closure.foo.bar};
}
Where a closure is not required, a frame is instead generated.

```c
void bar()
{
    int add = 2;
    scope dg = (int a) => a + add;
    assert(dg(5) == 7);
}
```
frame.bar.__lambda1 (void *this)
{
    return <retval> = a + ((FRAME.frame.bar *) this)->add;
}

frame.bar ()
{
    struct dg;
    int add [value-expr: (&__frame)->add];
    struct FRAME.frame.bar __frame;

    __frame.__chain = 0B;
    (&__frame)->add = 2;
    dg = {.object=&__frame, .func=frame.bar.__lambda1};
    if (dg.func (dg.object, 5) == 7)
    {
        0
    }
    else
    {
        _d_assert ({.length=6, .ptr="test.d"}, 7);
    }
}
• **gcc.builtins** gives access to built-ins provided by the GCC backend.

```c
import gcc.builtins;

void test()
{
    real r = 0.5 * __builtin_sqrtl(real.min_normal);

    __builtin_printf("Hello World!\n");
}
```
Generates GENERIC Code

- Allows many C library calls to be optimised in certain cases.

```c
builtins.test ()
{
    real r;

    r = 9.16801933777423582810706196024241582978182485679283618642e-2467;

    __builtin.puts ("Hello World!");
}
```
## Built-in Types

- Defines aliases to internal types.

```c
__builtin_va_list;  // Target C va_list type.
__builtin_clong;   // Target C long int type.
__builtin_culong;  // Target C long unsigned int type.
__builtin_machine_byte; // Signed type whose size is equal to sizeof(unit).
__builtin_machine_ubyte;  // Unsigned variant.
__builtin_machine_int;  // Signed type whose size is equal to sizeof(word).
__builtin_machine_uint;  // Unsigned variant.
__builtin_pointer_int;  // Signed type whose size is equal to sizeof(pointer).
__builtin_pointer_uint;  // Unsigned variant.
__builtin_unwind_int;   // Target C _Unwind_Sword type, for EH.
__builtin_unwind_uint;  // Target C _Unwind_Word type, for EH.
```
DMD has several intrinsics to the compiler.

```d
import core.bitop;
import core.math;

void main()
{
    long l;
    l = rndtol (4.5);

    size_t[2] a = [2, 256];
    btc(a.ptr, 35);
}
```
• **core.math** intrinsics are mapped to GCC builtin-ins.
• **core.bitop** instrinsics are expanded with inlined generated code.

```c
int main()
{
    int D.2001;
    ulong a[2];
    long l;
    l = 0;
    l = (long) __builtin_llroundl (4.5e+0);
    a[0] = 2;
    a[1] = 256;
    D.2001 = (*(ulong *) &a & 34359738368) != 0 ? -1 : 0;
    *(ulong *) &a = *(ulong *) &a ^ 34359738368;
    return <retval> = 0;
}
```
Many functions defined in `core.stdc` are mapped to GCC built-ins.

Functions recognised as a GCC built-in can be optimised.

Can be turned off with `-fno-built-in` switch.
import core.stdc.stdio;
import core.stdc.math;

void test()
{
    real r = powl(3, 3);

    if (r == 27.0)
        printf("Match!\n");
}

intrinsic.test()
{
    real r;

    r = 2.7e+1;
    {
        if (r == 2.7e+1)
            {
                __builtin_puts("Match!");
            }
    }
}
Variadic Functions

- The va_list type has an exclusive meaning in the compiler.

- Matches the C ABI, type is not a void*.

- Defined in `gcc.builtins`, then an alias to the type in `core.stdcc.stdarg`.

- Special va functions expanded at compile-time.
Variadic Functions

```c
import core.std.c.stdarg;

void variadic(...) {
    auto a1 = va_arg!(int)(_argptr);
    auto a2 = va_arg!(double)(_argptr);
    auto a3 = va_arg!(int[2])(_argptr);
    auto a4 = va_arg!(string)(_argptr);
}
```
valist.variadic (struct TypeInfo_Tuple & _arguments_typeinfo)
{
    struct _argptr[1];
    struct a4;
    int a3[2];
    double a2;
    int a1;
    struct _arguments;

    __builtin_va_start (&_argptr, _arguments_typeinfo);
    try
    {
        _arguments = _arguments_typeinfo->elements;
        a1 = VA_ARG_EXPR <_argptr>;
        a2 = VA_ARG_EXPR <_argptr>;
        a3 = VA_ARG_EXPR <_argptr>;
        a4 = VA_ARG_EXPR <_argptr>;
    }
    finally
    {
        __builtin_va_end (&_argptr);
    }
}
GCC Attributes

- Used to be accessible via pragmas in the language.
- Now uses UDA syntax that gets handled by `gcc.attributes`.

```c
import gcc.attributes;
import gcc.builtins;

@attribute("noreturn")
void die()
{
    __builtin_unreachable();
}

@attribute("forceinline")
int multiply(int a, int b)
{
    return a * b;
}
```
Attributes can also be applied to types.

```java
import gcc.attributes;

@attribute("aligned")
struct A
{
    char c;
    int i;
}

@attribute("unused") int unused_var;
```

As of writing, none of these type attributes are implemented in GDC.
GDC implements a variant of GCC Extended Assembly.

Extended assembly allows you to optionally specify the operands.

```asm
asm {
    "rdtsc"
    : /* output operands */
    : /* input operands */
    : /* list of clobbered registers */;
}
```
static uint getIeeeFlags()
{
    asm
    {
        fstsw AX;
        and EAX, 0x03D;
    }

    uint result;
    asm
    {
        "fstsw %%ax;
        andl $0x03D, %%eax"
        : "=a" result;
    }
    return result;
}
Benefits of Extended Assembly

- It is available on nearly all targets.

- Instruction templates can be generated through CTFE string constants.

- Does not prevent a function from being inlined.

- Can have some common optimisations applied to them, such as DCE.
Future Plans
Compiler: Short Term

- Find a workable solution for TLS support.
- Better support for LTO.
- Correct generation of D Thunks.
- Implement D Front-End in D.

- Integration of DFE into GCC garbage collector.
Add support for label operands in Extended Assembly.

```c
int frob(int x)
{
    int y;
    asm {
        "frob %%r5, %1;
        jc %l[Lerror];
        mov (%2), %%r5"
    : "r"(x), "r"(&y)
    : "r5", "memory"
    : Lerror;
    }
    return y;

Lerror:
    return -1;
}
```
- Implement Exception Chaining.

- Conversion of D IASM to Extended Assembly.

- Finish off port of ARM.

- Fix D GC runtime for TLS support.
http://gdcproject.org

http://bugzilla.gdcproject.org

https://github.com/D-Programming-GDC/GDC

ibuclaw@gdcproject.org
Questions?