The D Programming Language

by Walter Bright

A presentation by Daniel Korsgaard

Walter Bright

- First native C++ compiler (Zortech C++)
- Primary author of Sun's ECMA 262 Script engine.
- Lots of other compilers.
- Graduated from Caltech in 1979 with a BS in mechanical engineering.
- Author of D

What is D?

- Multiparadigm; Object oriented, imperative and metaprogramming.
- A general purpose systems and applications programming language.
- Unification of many modern programming languages.
- A practical language.
- Easy to learn.

Why D?

- C/C++ has plenty of stuff to improve upon
- Cannot keep extending the same language endlessly while retaining backwards compatibility.
- Lets reflect on years of experience and take everything we know and combine it into a new modern language.

D is generally very similar to C

```d
import std.stdio;

int main()
{
    writef("Hello World!");
    return 0;
}
```

D

```c
#include <stdio.h>

int main()
{
    printf("Hello World!");
    return 0;
}
```

C

- The C style of programming is widely used

Prominent differences to C/C++

- Module system (better code generation)
- Garbage collected (by default)
- First class arrays (data pointer and length)
- Array slicing (memory saver)
- Associative arrays (ease)
- Pointers are abstracted away (but still there)
- Intuitive type declaration
- Fixed size types (very portable)

A trim left example

```d
char[] ltrim(char[] str) {
    uint cut;

    foreach (uint i, char c; str)
        if (c != ' ') {
            cut = i;
            break;
        }

    return str[cut .. str.length];
}
```

D

```c
char *ltrim(char *str) {
    int len = strlen(str);
    int cut = 0;
    for (; cut < len; cut++)
        if (str[cut] != ' ')
            break;

    len = len - cut + 1;
    char *r = (char)malloc(len);
    return memcpy(r, &str[cut]);
}
```

C

- No visible pointers (but they are accessible if needed)

A trim left example

```d
char[] ltrim(char[] str) {
    uint cut;

    foreach (uint i, char c; str)
        if (c != ' ') {
            cut = i;
            break;
        }

    return str[cut .. str.length];
}
```

D

```c
char *ltrim(char *str) {
    int len = strlen(str);
    int cut = 0;
    for (; cut < len; cut++)
        if (str[cut] != ' ')
            break;

    len = len - cut + 1;
    char *r = (char)malloc(len);
    return memcpy(r, &str[cut]);
}
```

C

- Higher level language constructs

Arrays

```
int[] foo = [6, 3, 5, 8, 5];
int len = foo.length;
int* ptr = foo.ptr;

int[] slice = foo[1 .. 3];
// slice contains [3, 5]

writef(foo.reverse);
// prints ”[5, 8, 5, 3, 6]”
writef(foo.sort);
// prints ”[3, 5, 5, 6, 8]”
foo[10]++; // runtime error!
```

- First class values
- Slicing!
- Built in sorting
- Build in reversing
- Explicit length
- Bounds checked

Associative arrays

```
char[] [ char[] ] map;

map = [
    ”key”[] : ”value”[],
    ”more” : ”features!”,
];

char[]* r = (”key” in map);
if (r !is null) *r = ”new val”;

writefln(”%s”, map[”key”]);
// prints ”new val”
```

- Intuitive declaration
- Nice to have for quick and dirty programming.
- Hash table
- Initializers

More useful features

- String comparison and switch
- Resource Acquisition Is Initialization
- Contract programming
- Unit testing
- Conditional compilation (version debugging)
- Templates

String comparisons

```
char[] str = ”foo”;
switch (str) {
    case ”foo”:
        foo(); break;
    case ”bar”:
        bar(); break;
}
if (str == ”baz”)
    baz();
```

- Simple
- Convenient
- Does your laundry

RAII

```
void main() {
    char[] path =
        ”example.txt”;
    scope File fh =
        new File(path);
    fh.write(”test”);
}
```

- Very efficient (stack allocated)
- Nice for short term use of resources.
- All types can be stack allocated.
- Very efficient
- Destructors get called at end of scope

Contract programming

```
char[] ltrim(char[] str)
in {
    assert (str !is null, ”Fail in!”);
}
out (r) {
    if (r.length > 0)
        assert (r[0] != ’ ’, ”Fail out!”);
}
body {
    uint cut;
    …
    return str[cut .. str.length];
}
```

- Calling ”ltrim(null);” will give error: ”Error: AssertError Failure test.d(17) Fail in!”
- Calling a buggy ltrim will give error: ”Error: AssertError Failure test.d(21) Fail out!”
- Potentially faster code

Unittests

```
unittest {
    writefln(”testing ltrim”);
    assert (ltrim(”h”) == ”h”);
    assert (ltrim(” h”) == ”h”);
    assert (ltrim(”  h”) == ”h”);
    assert (ltrim(”h ”) == ”h ”);
    assert (ltrim(” h ”) == ”h ”);
    assert (ltrim(”      ”) == ””);
    writefln(”done testing”);
}
```

- Use switch to compile in.
- Serves both as documentation and insure that the code works as the original author intended.
- Puts “cool” back into testing.
- Can be used as specification for a lib

Conditional compilation

```
version (Pro) {
    very_fast_sorting();
} else version (Demo) {
    slow_sorting();
}

debug (3) {
    writefln(”foo called!”);
}
```

- Specialized solution instead of macros
- Again, intuitive!
- Supports levels of debugging and versioning.
- Built in predefined versions, windows, linux, x86, x86_64, …

Templates

```
template factorial(int n) {
    static if (n == 1)
        const factorial = 1;
    else
        const factorial =
            n * factorial!(n - 1);
}

void main() {
    writefln("%d", factorial!(4));
}
```

D

- Goodbye to angle brackets
- Much simpler lookup rules compared to C++
- All of the features of C++ templates
- Everything can be templated!
- Very powerful

- Compile-time ray-tracer!

Compiletime Factorial in C++

```
template<int n> class factorial {
    public:
        enum { result = n * factorial<n - 1>::result };
};
template<> class factorial<1> {
    public:
        enum { result = 1 };
};

void main() {
    printf("%d\n", factorial<4>::result);
}
```

Final words

- Contrary to other languages in this course D is actually practical beyond toy examples.
- C++ good features but ugly and complex code!
- C# more good features, too far from the hardware, and too object oriented.
- D very good combination of good features of many modern languages, while retaining flexibility.
- D does lack a bit in the reflection compartment
- D’s current feature set has much more interesting stuff going on.