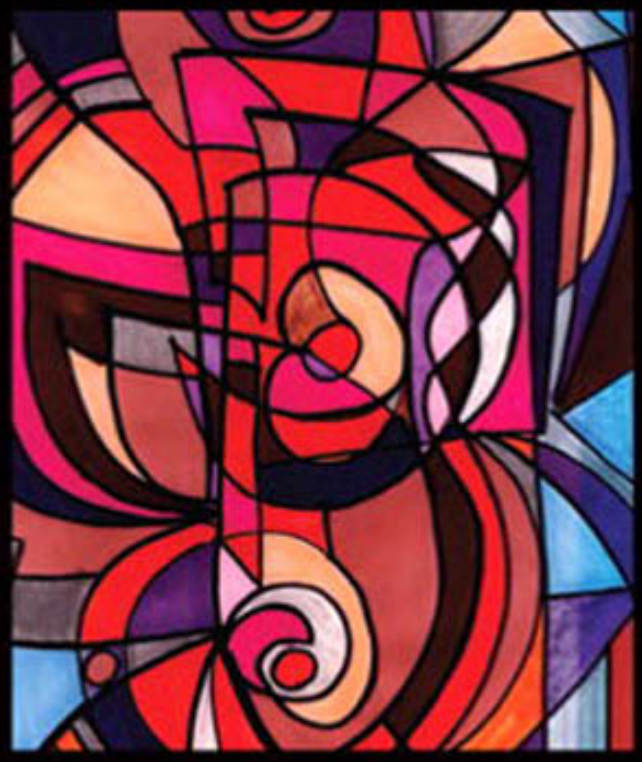
FOREWORDS BY WALTER BRIGHT AND SCOTT MEYERS



The D Programming Language

Andrei Alexandrescu

The D Programming Language

1. A non-leading byte is never equal to a leading byte.
2. The first byte unambiguously determines the length of an encoding.

The first property is crucial because it enables two important applications. One is simple synchronization—if you pick up a UTF-8 transmission somewhere in midstream, you can easily figure out where the next code point starts: just look for the next byte with anything but `10` as its most significant bits. The other application is backward iteration—it is easy to go backward in a UTF-8 string without ever getting confused. Backward iteration opens UTF-8 strings to a host of algorithms (e.g., finding the last occurrence of a string in another can be implemented efficiently). The second property is not essential but simplifies and accelerates string processing.

Ideally, frequent code points should have small values and infrequent ones should have large values. If that condition is fulfilled, UTF-8 acts as a good statistical encoder by encoding more frequent symbols in fewer bits. This is certainly the case for Latin-derived languages, where most code units fit in 1 byte and the occasional accented characters fit in 2.

UTF-16 is also a variable-length encoding but uses a different (and arguably less elegant) approach to encoding. Code points between `0` and `0xFFFF` are encoded as a sole 16-bit code unit and code points between `0x10000` and `0x10FFFF` are represented by a pair in which the first code unit is in the range `0xD800` through `0xDBFF` and the second code unit is in the range `0xDC00` through `0xDFFF`. To support this encoding, Unicode allocates no valid characters to numbers in the range `0xD800` through `0xDBFF`. The two ranges are called *high surrogate area* and *low surrogate area*, respectively.

One criticism commonly leveled against UTF-16 is that it makes the statistically rare cases also the most complicated and the ones deserving the most scrutiny. Most—but alas, not all—Unicode characters (the so-called Basic Multilingual Plane) *do* fit in one UTF-16 code unit, and therefore a lot of UTF-16 code tacitly assumes one code unit per character and is effectively untested for surrogate pairs. To further the confusion, some languages initially centered their string support around UCS-2, a precursor of UTF-16 with exactly 16 bits per code point, to later add UTF-16 support, subtly obsoleting older code that relied on a one-to-one mapping between characters and codes.

Finally, UTF-32 uses 32 bits per code unit, which allows a true one-to-one mapping of code points to code units. This means UTF-32 is the simplest and easiest-to-use representation, but it's also the most space-consuming. A common recommendation is to use UTF-8 for storage and UTF-32 temporarily during processing if necessary.

4.5.3 Character Types

D defines three character types: `char`, `wchar`, and `dchar`, representing code units for UTF-8, UTF-16, and UTF-32, respectively. Their `.init` values are intentionally invalid encodings: `char.init` is `0xFF`, `wchar.init` is `0xFFFF`, and `dchar.init` is `0x0000FFFF`.

Table 4.2 on page 119 clarifies that `0xFF` may not be part of any valid UTF-8 encoding, and also Unicode deliberately assigns no valid code point for `0xFFFF`.

Used individually, the three character types mostly behave like unsigned integers and can occasionally be used to store invalid UTF code points (the compiler does not enforce valid encodings throughout), but the intended meaning of `char`, `wchar`, and `dchar` is as UTF code points. For general 8-, 16-, and 32-bit unsigned integers, or for using encodings other than UTF, it's best to use `ubyte`, `ushort`, and `uint`, respectively. For example, if you want to use pre-Unicode 8-bit code pages, you may want to use `ubyte`, not `char`, as your building block.

4.5.4 Arrays of Characters + Benefits = Strings

When assembling any of the character types in an array—as in `char[]`, `wchar[]`, or `dchar[]`—the compiler and the runtime support library "understand" that you are working with UTF-encoded Unicode strings. Consequently, arrays of characters enjoy the power and versatility of general arrays, plus a few extra goodies as Unicode denizens.

In fact, D already defines three string types corresponding to the three character widths: `string`, `wstring`, and `dstring`. They are not special types at all; in fact, they are aliases for character array types, with a twist: the character type is adorned with the `immutable` qualifier to disallow arbitrary changes of individual characters in strings. For example, type `string` is a synonym for the more verbose type `immutable(char)[]`. We won't get to discussing type qualifiers such as `immutable` until Chapter 8, but for strings of all widths the effect of `immutable` is very simple: a `string`, aka an `immutable(char)[]`, is just like a `char[]` (and a `wstring` is just like a `wchar[]`, etc.), except you can't assign new values to individual characters in the string:

```
string a = "hello";
char h = a[0];       // Fine
a[0] = 'H';          // Error!
                     // Cannot assign to immutable(char)!
```

To change one individual character in a `string`, you need to create another `string` via concatenation:

```
string a = "hello";
a = 'H' ~ a[1 .. $];  // Fine, makes a == "Hello"
```

Why such a decision? After all, in the case above it's quite a waste to allocate a whole new `string` (recall from § 4.1.6 on page 100 that ~ always allocates a new array) instead of just modifying the existing one. There are, however, a few good reasons for disallowing modification of individual characters in strings. One reason is that `immutable` simplifies situations when `string`, `wstring`, and `dstring` objects are copied and then changed. Effectively `immutable` ensures no undue aliasing between strings. Consider:

```
string a = "hello";
string b = a;          // b is also "hello"
string c = b[0 .. 4];  // c is "hell"
// If this were allowed, it would change a, b, and c
// a[0] = 'H';
// The concatenation below leaves b and c unmodified
a = 'H' ~ a[1 .. $];
assert(a == "Hello" && b == "hello" && c == "hell");
```

With `immutable` characters, you know you can have several variables refer to the same string, without fearing that modifying one would also modify the others. Copying `string` objects is very cheap because it doesn't need to do any special copy management (such as eager copy or copy-on-write).

An equally strong reason for disallowing changes in strings at code unit level is that such changes don't make much sense anyway. Elements of a `string` are variable-length, and most of the time you want to replace logical characters (code points), not physical `chars` (code units), so you seldom want to do surgery on individual `chars`. It's much easier to write correct UTF code if you forgo individual `char` assignments and you focus instead on manipulating entire strings and fragments thereof. D's standard library sets the tone by fostering manipulation of strings as whole entities instead of focusing on indices and individual characters. However, UTF code is not trivially easy to write; for example, the concatenation `'H' ~ a[1 .. $]` above has a bug in the general case because it assumes that the first code point in `a` has exactly 1 byte. The correct way to go about it is

```
a = 'H' ~ a[stride(a, 0) .. $];
```

The function `stride`, found in the standard library module `std.utf`, returns the length of the code starting at a specified position in a string. (To use `stride` and related library artifacts, insert the line

```
import std.utf;
```

near the top of your program.) In our case, the call `stride(a, 0)` returns the length of the encoding for the first character (aka code point) in `a`, which we pass to select the offset marking the beginning of the second character.

A very visible artifact of the language's support for Unicode can be found in string literals, which we've already looked at (§ 2.2.5 on page 35). D string literals understand Unicode code points and automatically encode them appropriately for whichever encoding scheme you choose. For example:

```
import std.stdio;
```

```
void main() {
   string a = "No matter how you put it, a \u03bb costs \u20AC20.";
   wstring b = "No matter how you put it, a \u03bb costs \u20AC20.";
   dstring c = "No matter how you put it, a \u03bb costs \u20AC20.";
   writeln(a, '\n', b, '\n', c);
}
```

Although the internal representations of `a`, `b`, and `c` are very different, you don't need to worry about that because you express the literal in an abstract way by using code points. The compiler takes care of all encoding details, such that in the end the program prints three lines containing the same exact text:

```
No matter how you put it, a λ costs €20.
```

The encoding of the literal is determined by the context in which the literal occurs. In the cases above, the compiler has the literal morph without any runtime processing into the encodings UTF-8, UTF-16, and UTF-32 (corresponding to types `string`, `wstring`, and `dstring`), in spite of it being spelled the exact same way throughout. If the requested literal encoding is ambiguous, suffixing the literal with one of `c`, `w`, or `d` (something `"like that"d`) forces the encoding of the string to UTF-8, UTF-16, and UTF-32, respectively (refer to § 2.2.5.2 on page 37).

4.5.4.1 `foreach` with Strings

If you iterate a string `str` of any width like this:

```
foreach (c; str) {
   ... // Use c
}
```

then `c` will iterate every *code unit* of `str`. For example, if `str` is an array of `char` (`immutable` or not), `c` takes type `char`. This is expected from the general behavior of `foreach` with arrays but is sometimes undesirable for strings. For example, let's print each character of a `string` enclosed in square brackets:

```
void main() {
   string str = "Hall\u00E5, V\u00E4rld!";
   foreach (c; str) {
      write('[', c, ']');
   }
   writeln();
}
```

The program above ungainly prints

```
[H][a][l][l][?][?][,][ ][V][?][?][r][l][d][!]
```

The reverse video `?` (which may vary depending on system and font used) is the console's mute way of protesting against seeing an invalid UTF code. Of course, trying to print alone a `char` that would make sense only in combination with other `char`s is bound to fail.

The interesting part starts when you specify a different character type for `c`. For example, specify `dchar` for `c`:

```
... as above, just add "dchar" ...
foreach (dchar c; str) {
   write('[', c, ']');
}
```

In this case, the compiler automatically inserts code for transcoding on the fly each code unit in `str` in the representation dictated by `c`'s type. The loop above prints

```
[H][a][l][l][å][,][ ][v][ä][r][d][!]
```

which indicates that the double-byte characters `å` and `ä` were converted correctly to one `dchar` each and subsequently printed correctly. The same exact result would be printed if `c` had type `wchar` because the two non-ASCII characters used fit in one UTF-16 unit each, but not in the most general case (surrogate pairs would be wrongly processed). To be on the safe side, it is of course best to use `dchar` with loops over strings.

In the case above, the transcoding performed by `foreach` went from a narrow to a wide representation, but it could go either way. For example, you could start with a `dstring` and iterate it one (encoded) `char` at a time.

4.6 Arrays' Maverick Cousin: The Pointer

An array object tracks a chunk of typed objects in memory by storing the lower and upper bound. A pointer is "half" an array—it tracks only one object. As such, the pointer does not have information on whether the chunk starts and ends. If you have that information from the outside, you can use it to move the pointer around and make it point to neighboring elements.

A pointer to an object of type `T` is denoted as type `T*`, with the default value `null` (i.e., a pointer that points to no actual object). To make a pointer point to an object, use the address-of operator `&`, and to use that object, use the dereference operator `*` (§ 2.3.6.2 on page 52). For example:

```
int x = 42;
int* p = &x;     // Take the address of x
*p = 10;         // Using *p is the same as using x
++*p;            // Regular operators also apply
```

```
assert(x == 11); // x was modified through p
```

Pointers allow arithmetic that makes them apt as cursors inside arrays. Incrementing a pointer makes it point to the next element of the array; decrementing it moves it to the previous element. Adding an integer `n` to a pointer yields a pointer to an object situated `n` positions away in the array, to the right if `n` is positive and to the left if `n` is negative. To simplify indexed operations, `p[n]` is equivalent to `*(p + n)`. Finally, taking the difference between two pointers `p2 - p1` yields an integral `n` such that `p1 + n == p2`.

You can fetch the address of the first element of an array with `a.ptr`. It follows that a pointer to the last element of a non-empty array `arr` can be obtained with `arr.ptr + arr.length - 1`, and a pointer just past the last element with `arr.ptr + arr.length`. To exemplify all of the above:

```
auto arr = [ 5, 10, 20, 30 ];
auto p = arr.ptr;
assert(*p == 5);
++p;
assert(*p == 10);
++*p;
assert(*p == 11);
p += 2;
assert(*p == 30);
assert(p - arr.ptr == 3);
```

Careful, however: unless you have access to array bounds information from outside the pointer, things could go awry very easily. All pointer operations go completely unchecked—the implementation of the pointer is just a word-long memory address and the corresponding arithmetic just blindly does what you ask. That makes pointers blazingly fast and also appallingly ignorant. Pointers aren't even smart enough to realize they are pointing at individual objects (as opposed to pointing inside arrays):

```
auto x = 10;
auto y = &x;
++y;         // Huh?
```

Pointers also don't know when they fall off the limits of arrays:

```
auto x = [ 10, 20 ];
auto y = x.ptr;
y += 100;              // Huh?
*y = 0xdeadbeef;       // Russian roulette
```

Writing through a pointer that doesn't point to valid data is essentially playing Russian roulette with your program's integrity: the writes could land anywhere, stomping