Appendix C. A Unicode Primer

This isn't a complete or comprehensive introduction to Unicode; it's just enough for you to understand the parts of Unicode that we present in this book. Unicode is tricky not only because it's a new way to think about strings, with lots of adjusted vocabulary, but also because computer languages in general have implemented it so poorly. Each version since v5.6 has brought Perl closer to full compliance. Perl has, arguably, the best Unicode support that you will find, though.

Unicode

The Universal Character Set (UCS) is an abstract mapping of *characters* to *code points*. It has nothing to do with a particular representation in memory, which means we can agree on at least one way to talk about characters no matter which platform we're on. An *encoding* turns the code points into a particular representation in memory, taking the abstract mapping and representing it physically within a computer. You probably think of this storage in terms of bytes, although when talking about Unicode, we use the term *octets* (see Figure C-1). Different encodings store the characters differently. To go the other way, interpreting the octets as characters, you *decode* them. You don't have to worry too much about these because Perl can handle most of the details for you.

When we talk about a code point, we specify its number in hexadecimal like so: (U+0158); that's the character *Ř*. Code points also have names, and that code point is "LATIN CAPITAL LETTER R WITH CARON." Not only that, but code points know certain things about themselves. They know if they are an uppercase or lowercase character, a letter or digit or whitespace, and so on. They know what their uppercase, title case, or lowercase partner is, if appropriate. This means that not only can we work with the particular characters, but we now have a way to talk about *types* of characters. All of this is defined in Unicode datafiles that come with *perl*.

Look for a *unicore* directory in your Perl library directory; that's how Perl knows everything it needs to know about characters.

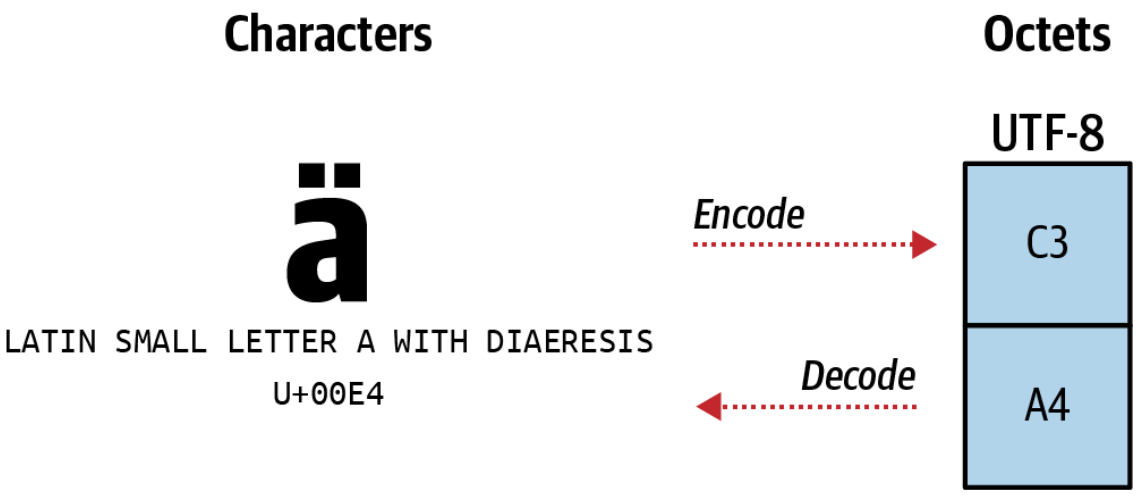

Figure C-1. The code point of a character is not its storage. The encoding transforms characters into storage.

UTF-8 and Friends

The preferred encoding in Perl is UTF-8, which is short for UCS Transformation Format 8-bit. Rob Pike and Ken Thompson defined this encoding one night on the back of a paper placemat in a New Jersey diner. It's just one possible encoding, although a very popular one since it doesn't have the drawbacks of some other encodings. If you're using Windows, you're likely to run into UTF-16. We don't have anything nice to say about that encoding, so we'll just keep quiet like our mothers told us.

NOTE

Read about the invention of UTF-8 from Rob Pike himself.

Getting Everyone to Agree

Getting everything set up to use Unicode can be frustrating because every part of the system needs to know which encoding to expect so it can display it properly. Mess up on any part of that and you might see gibberish, with no clue which part isn't working correctly. If your program outputs

UTF-8, your terminal needs to know that so it displays the characters correctly. If you input UTF-8, your Perl program needs to know that so it interprets the input strings correctly. If you put data into a database, the database server needs to store it correctly and return it correctly. You have to set up your editor to save your source in UTF-8 if you want *perl* to interpret your typing as UTF-8.

We don't know which terminal you are using and we're not going to list instructions for every (or any) terminal here. For modern terminal programs, you should find a setting in the preferences or properties for the encoding.

Beyond the encoding, various programs need to know how to output the encoding that you want. Some look at the `LC_*` environment variables and some have their own:

```
LESSCHARSET=utf-8
LC_ALL=en_US.UTF-8
```

If something is not displaying correctly through your pager (i.e., less, more, type), read their documentation to see what they expect you to set to give encoding hints.

Fancy Characters

Thinking in Unicode requires a different mindset if you are used to ASCII. For instance, what's the difference between *é* and *é*? You probably can't tell just by looking at those, and even if you have the digital version of this book, the publication process might have "fixed" the difference. You might not even believe us that there is a difference, but there is. The first one is a single character but the second one is two characters. How can that be? To humans, those are the same thing. To us, they are the same *grapheme* (or *glyph*) because the idea is the same no matter how the computer deals with either of them. We mostly care about the end result (the grapheme) since that's what imparts information to our readers.

Before Unicode, common character sets defined characters such as *é* as an atom, or single entity. That's the first of our examples in the previous paragraph (just trust us). However, Unicode also introduces the idea of *mark* characters—the accents and other flourishes and annotations that combine with another character (called *nonmarks*). That second *é* is actually the nonmark character *e* (U+0065, LATIN SMALL LETTER E) and the mark character ´ (U+0301, COMBINING ACUTE ACCENT) that is the pointy part over the letter. These two characters together make up the grapheme. Indeed, this is why you should stop calling the overall representation a character and call it a grapheme instead. One or more characters can make up the final grapheme. It's a bit pedantic, but it makes it much easier to discuss Unicode without going insane.

If the world were starting fresh, Unicode probably wouldn't have to deal with the single-character version of *é*, but the single-character version exists historically, so Unicode does handle it to be somewhat backward compatible and friendly with the text that's already out there. Unicode code points have the same ordinal values for the ASCII and Latin-1 encodings, which are all the code points from 0 to 255. That way, treating your ASCII strings as UTF-8 should work out just fine (but not UTF-16, where every character takes up at least two bytes).

The single-character version of *é* is a *composed* character because it represents two (or more) characters as one code point. It composes the nonmark and mark into a single character (U+00E9, LATIN SMALL LETTER E WITH ACUTE) that has its own code point. The alternative is the *decomposed* version that uses two characters.

So, why do you care? How can you properly sort text if what you think of as the same thing is actually different characters? Perl's `sort` cares about characters, not graphemes, so the strings `"\x{E9}"` and `"\x{65}\x{301}"`, which are both logically *é*, do not sort to the same position. Before you sort these strings, you want to ensure that both *é*'s sort next to each other no matter how you represent them. Computers don't sort in the same way that humans want to sort items. You don't care about composed or decomposed characters. We'll show you the solution in a moment, and you should check Chapter 14.

Using Unicode in Your Source

If you want to have literal UTF-8 characters in your source code, you need to tell *perl* to read your source as UTF-8. You do that with the `utf8` pragma, whose only job is to tell *perl* how to interpret your source. This example has Unicode characters in a string:

```
use utf8;

my $string = "Here is my 🐻 résumé";
```

You can also use some characters in variable and subroutine names:

```
use utf8;

my %résumés = (
    Fred => 'fred.doc',
    ...
    );

sub π () { 3.14159 }
```

The only job of the `utf8` pragma is to tell *perl* to interpret your source code as UTF-8. It doesn't do anything else for you. As you decide to work with Unicode, it's a good idea to always include this pragma in your source unless you have a good reason not to.

NOTE

Typing characters you don't see on your keyboard may be difficult. Services such as r12a's Unicode code converter and UniView 9.0.0 or a program such as UnicodeChecker can help.

Fancier Characters

It gets worse, though, although not as many of you probably care about this one. What's the difference between *fi* and *ﬁ*? Unless the typesetter

“optimized” this, the first one has the *f* and the *i* separated while the second one combines those in a *ligature*, which generally sets the graphemes in a way that makes it easier for people to read. The overhanging part of the *f* appears to impose on the personal space of the dot on the *i*, which is a bit ugly. We don’t actually read each letter in a word and instead recognize it as a whole; the ligature is a slight improvement in our pattern recognition. So typographers combine the two graphemes. You may have never noticed it, but you’ll find several examples in this paragraph, and you’ll find them often in typeset books (but usually not ebooks, which typically don’t care as much about looking nice).

NOTE

O’Reilly’s automated typesetting system doesn’t turn our *fi*’s into their ligature forms unless we type the ligatures ourselves. It’s probably a faster document workflow that way, even if we do have to shuffle some graphemes manually. Fingers crossed that it shows up the way we wanted!

The difference is similar to the composed and decomposed forms of *é*, but slightly different. The *é*’s were *canonically equivalent* because no matter which way you made it, the result was the same visual appearance and the same idea. The *fi* and *ﬁ* don’t have the same visual appearance, so they are merely *compatibility equivalent*. You don’t need to know too much about that other than knowing that you can decompose both canonically and compatibility equivalent forms to a common form that you can use to sort (Figure C-2). See Unicode Standard Annex #15, “Unicode Normalization Forms” for the gory details.

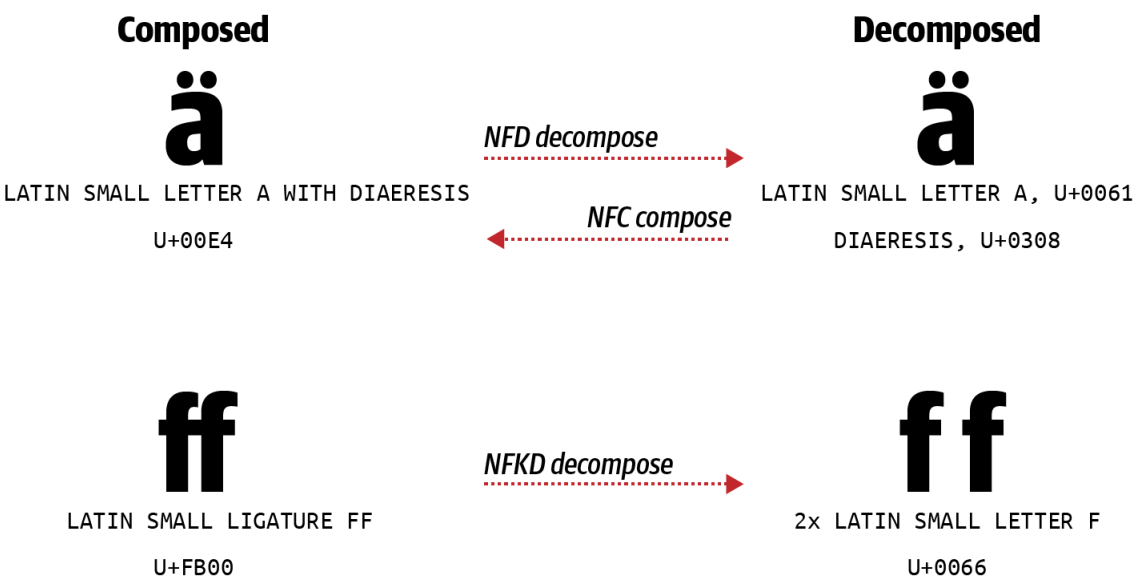

Figure C-2. You can decompose and recompose canonical equivalent forms, but you can only decompose compatible forms.

Suppose that you want to check if a string has an *é* or an *fi* and you don't care about which form it has. To do that, you decompose the strings to get them in a common form. To decompose Unicode strings, use the `Unicode::Normalize` module, which comes with Perl. It supplies two subroutines for decomposition. You use the `NFD` subroutine (*Normalization Form Decomposition*), which turns canonically equivalent forms into the same decomposed form. You use the `NFKD` subroutine (*Normalization Form Kompatibility Decomposition*) to convert to compatible forms that represent the same thing but aren't the same thing (for example, *ss* for *ß*). This example has a string with composed characters that you decompose and match in various ways. The "oops" messages shouldn't print, while the "yay" messages should:

```
use utf8;
use Unicode::Normalize;

# U+FB01        - fi ligature
# U+0065 U+0301 - decomposed é
# U+00E9        - composed é

binmode STDOUT, ':utf8';

my $string =
    "Can you \x{FB01}nd my r\x{E9}sum\x{E9}?";

if( $string =~ /\x{65}\x{301}/ ) {
```

```
    print "Oops! Matched a decomposed é\n";
}
if( $string =~ /\x{E9}/ ) {
    print "Yay! Matched a composed é\n";
}

my $nfd  = NFD( $string );
if( $nfd =~ /\x{E9}/ ) {
    print "Oops! Matched a composed é\n";
}
if( $nfd =~ /fi/ ) {
    print "Oops! Matched a decomposed fi\n";
}

my $nfkd = NFKD( $string );
if( $string =~ /fi/ ) {
    print "Oops! Matched a decomposed fi\n";
}
if( $nfkd =~ /fi/ ) {
    print "Yay! Matched a decomposed fi\n";
}
if( $nfkd =~ /\x{65}\x{301}/ ) {
    print "Yay! Matched a decomposed é\n";
}
```

As you can see, the NFKD forms always match the decompositions because `NFKD()` decomposes both canonical and compatible equivalents. The NFD forms miss the compatible equivalents:

```
Yay! Matched a composed é
Yay! Matched a decomposed fi
Yay! Matched a decomposed é
```

There's a caution here, though: you can decompose and recompose canonical forms, but you cannot necessarily recompose compatible forms. If you decompose the ligature *fi*, you get the separate graphemes `f` and `i`. The recomposer has no way to know if those came from a ligature or started separately. (This is why we're ignoring NFC and NFKC. Those forms decompose then recompose, but NFKC can't necessarily recompose

to the original form.) Again, that's the difference in canonical and compatible forms: the canonical forms look the same either way.

Dealing with Unicode in Perl

This section is a quick summary of the most common ways you'll incorporate Unicode into your Perl programs. This is not a definitive guide, and even for the things we do show there are some details that we ignore. It's a big subject, and we don't want to scare you off. Learn a little at first (this appendix), but when you run into problems, reach for the detailed documentation we list at the end of the appendix.

Fancier Characters by Name

Unicode characters also have names. If you can't easily type the character with your keyboard and you can't easily remember the code points, you can use its name (although it is a lot more typing). The `charnames` pragma, which comes with Perl, gives you access to those names. Put the name inside `\N{...}` in a double-quotish context:

```
my $string = "\N{THAI CHARACTER KHOMUT}"; # U+0E5B
```

Note that the pattern portions of the match and substitution operators are also double-quoted context, but there's also a character class shortcut `\N` that means "not a newline" (see Chapter 8). It usually works out just fine because there are only some weird cases where Perl might get confused. For a detailed discussion of the `\N` problem, see the blog post "Use the /N regex character class to get 'not a newline'" for more information.

Reading from STDIN or Writing to STDOUT or STDERR

At the lowest level, your input and output is just octets. Your program needs to know how to decode or encode them. We've mostly covered this in Chapter 5, but here's a summary.

You have two ways to use a particular encoding with a filehandle. The first one uses `binmode`:

```
binmode STDOUT, ':encoding(UTF-8)';
binmode $fh, ':encoding(UTF-16LE)';
```

You can also specify the encoding when you open the filehandle:

```
open my $fh, '>:encoding(UTF-8)', $filename;
```

If you want to set the encoding for all filehandles that you will open, you can use the `open` pragma. You can affect all input or all output filehandles:

```
use open IN  => ':encoding(UTF-8)';
use open OUT => ':encoding(UTF-8)';
```

You can do both with a single pragma:

```
use open IN  => ":crlf", OUT => ":bytes";
```

If you want to use the same encoding for both input and output, you can set them at the same time, either using `IO` or omitting it:

```
use open IO  => ":encoding(iso-8859-1)";
use open ':encoding(UTF-8)';
```

Since the standard filehandles are already open, you can apply your previously stated encoding by using the `:std` subpragma:

```
use open ':std';
```

This last one has no effect unless you've already explicitly declared an encoding. In that case, add the encoding as the second import item:

```
use open qw(:std :encoding(UTF-8));
```

You can also set these on the command line with the `-C` switch, which will set the encodings on the standard filehandles according to the arguments you give to it:

```
I     1   STDIN is assumed to be in UTF-8
O     2   STDOUT will be in UTF-8
E     4   STDERR will be in UTF-8
S     7   I + O + E
i     8   UTF-8 is the default PerlIO layer for input streams
o    16   UTF-8 is the default PerlIO layer for output streams
D    24   i + o
```

See the perlrun documentation for more information about command-line switches, including the details for `-C`.

Reading from and Writing to Files

We cover this in Chapter 5, but here's the summary. When you open a file, use the three-argument form and specify the encoding so you know exactly what you are getting:

```
open my( $read_fh ),   '<:encoding(UTF-8)',  $filename;
open my( $write_fh ),  '>:encoding(UTF-8)',  $file_name;
open my( $append_fh ), '>>:encoding(UTF-8)', $file_name;
```

Remember, though, that you don't get to pick the encoding of the input (at least not from inside your program). Don't choose an encoding for the input unless you are sure that's the encoding the input actually is. Notice that although you're really *decoding* input, you still use `:encoding`.

If you don't know what sort of input you'll get (and one of the Laws of Programming is that run enough times, you'll see every possible encoding), you can also just read the raw stream and guess the encoding, perhaps with `Encode::Guess`. There are many gotchas there, though, and we won't go into them here.

Once you get the data into your program, you don't need to worry about the encoding anymore. Perl stores it smartly and knows how to manipulate it. It's not until you want to store it in a file (or send it down a socket, and so on) that you need to encode it again.

Dealing with Command-Line Arguments

As we have said before, you need to be careful about the source of any data when you want to treat it as Unicode. The `@ARGV` array is a special case since it gets its values from the command line, and the command line uses the locale:

```
use I18N::Langinfo qw(langinfo CODESET);
use Encode qw(decode);

my $codeset = langinfo(CODESET);

foreach my $arg ( @ARGV ) {
    push @new_ARGV, decode $codeset, $arg;
}
```

Dealing with Databases

Our editor tells us that we are running out of space, and it's almost the end of the book! We don't have that much space to cover this topic, but that's OK because it's not really about Perl. Still, he's allowing us a couple of sentences. It's really too bad that we can't go into all the ways that database servers make life so hard, or how they all do it in different ways.

Eventually you'll want to store some of your information in a database. The most popular Perl module for database access, `DBI`, is Unicode-transparent, meaning it passes the data it gets directly to the database server without messing with it. Check its various drivers (for example, `DBD::mysql`) to see which driver-specific settings you'll need. You also have to set up your database server, schemas, tables, and columns correctly. Now you can see why we're glad we've run out of space!

Further Reading

There are several parts of the Perl documentation that will help you with the language-specific parts, including the perlunicode, perlunifaq, perluniintro, perluniprops, and perlunitut documentation. Don't forget to check the documentation for any of the Unicode modules that you use.

The official Unicode site has almost everything you'd ever want to know about Unicode, and is a good place to start.

There's also a Unicode chapter in *Effective Perl Programming* (Addison-Wesley), also by one of the authors of this book.