55.1.2 Byte and Character Semantics

Beginning with version 5.6, Perl uses logically-wide characters to represent strings internally.

In future, Perl-level operations will be expected to work with characters rather than bytes.

However, as an interim compatibility measure, Perl aims to provide a safe migration path from byte semantics to character semantics for programs. For operations where Perl can unambiguously decide that the input data are characters, Perl switches to character semantics. For operations where this determination cannot be made without additional information from the user, Perl decides in favor of compatibility and chooses to use byte semantics.

This behavior preserves compatibility with earlier versions of Perl, which allowed byte semantics in Perl operations only if none of the program's inputs were marked as being as source of Unicode character data. Such data may come from filehandles, from calls to external programs, from information provided by the system (such as %ENV), or from literals and constants in the source text.

The `bytes` pragma will always, regardless of platform, force byte semantics in a particular lexical scope. See *bytes*.

The `utf8` pragma is primarily a compatibility device that enables recognition of UTF-(8|EBCDIC) in literals encountered by the parser. Note that this pragma is only required while Perl defaults to byte semantics; when character semantics become the default, this pragma may become a no-op. See *utf8*.

Unless explicitly stated, Perl operators use character semantics for Unicode data and byte semantics for non-Unicode data. The decision to use character semantics is made transparently. If input data comes from a Unicode source–for example, if a character encoding layer is added to a filehandle or a literal Unicode string constant appears in a program–character semantics apply. Otherwise, byte semantics are in effect. The `bytes` pragma should be used to force byte semantics on Unicode data.

If strings operating under byte semantics and strings with Unicode character data are concatenated, the new string will be created by decoding the byte strings as *ISO 8859-1 (Latin-1)*, even if the old Unicode string used EBCDIC. This translation is done without regard to the system's native 8-bit encoding. To change this for systems with non-Latin-1 and non-EBCDIC native encodings, use the `encoding` pragma. See *encoding*.

Under character semantics, many operations that formerly operated on bytes now operate on characters. A character in Perl is logically just a number ranging from 0 to 2**31 or so. Larger characters may encode into longer sequences of bytes internally, but this internal detail is mostly hidden for Perl code. See *perluniintro* for more.

55.1.3 Effects of Character Semantics

Character semantics have the following effects:

- Strings–including hash keys–and regular expression patterns may contain characters that have an ordinal value larger than 255.

  If you use a Unicode editor to edit your program, Unicode characters may occur directly within the literal strings in one of the various Unicode encodings (UTF-8, UTF-EBCDIC, UCS-2, etc.), but will be recognized as such and converted to Perl's internal representation only if the appropriate *encoding* is specified.

  Unicode characters can also be added to a string by using the `\x{...}` notation. The Unicode code for the desired character, in hexadecimal, should be placed in the braces. For instance, a smiley face is `\x{263A}`. This encoding scheme only works for characters with a code of 0x100 or above.

  Additionally, if you

  ```
  use charnames ':full';
  ```

  you can use the `\N{...}` notation and put the official Unicode character name within the braces, such as `\N{WHITE SMILING FACE}`.

- If an appropriate *encoding* is specified, identifiers within the Perl script may contain Unicode alphanumeric characters, including ideographs. Perl does not currently attempt to canonicalize variable names.

- Regular expressions match characters instead of bytes. "." matches a character instead of a byte. The `\C` pattern is provided to force a match a single byte–a `char` in C, hence `\C`.