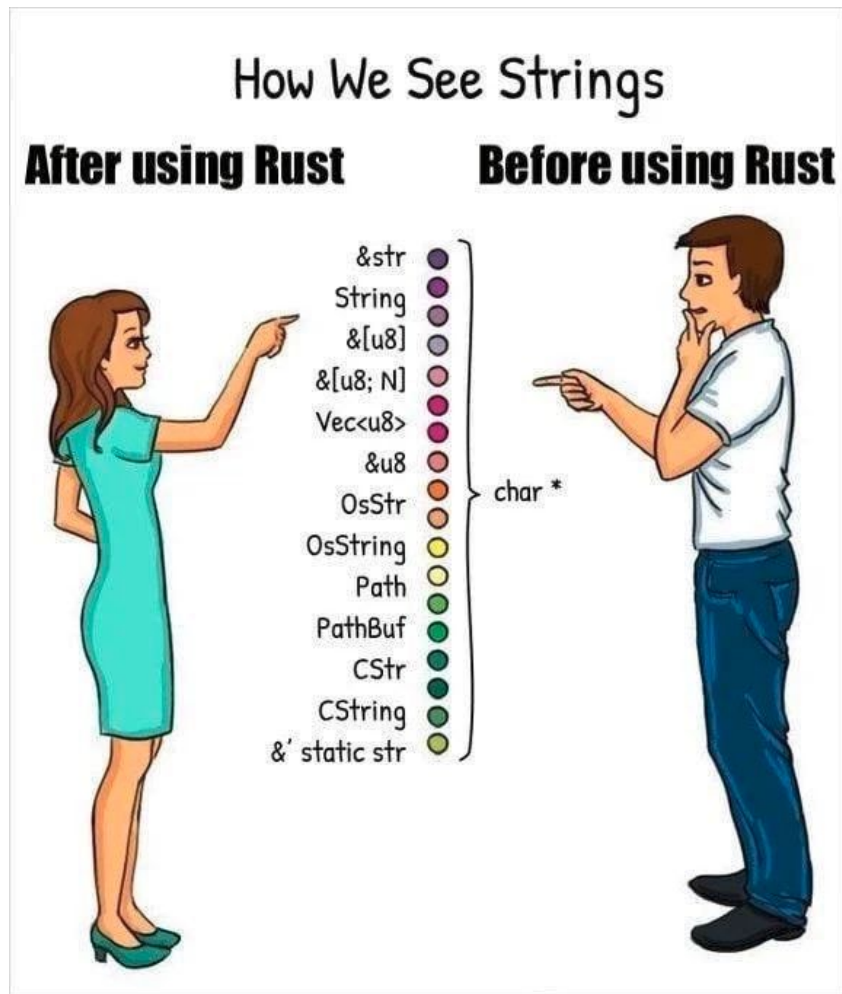
Lec 7: Strings

Jack Duvall

The Classic



Strings Are Important

- Many things can only be represented as strings
  - Names
  - Descriptions
  - Addresses
- Human-readable
- "Just" a list of characters!

How To Represent Strings?

1. How to encode length?
    - C, C++: implicitly with a NUL terminator
    - Every Other Language Since: explicitly, with a length field
2. What's the size of each character?
    - C, C++: at least 1 byte
    - Rust: exactly 1 byte
    - Java, C#: exactly 2 bytes
3. How are characters encoded?
    - C, C++: lmao what's that
    - Go, Python, Rust: UTF-8
    - Java, C#: UTF-16

Length Encoding

Implicit Length Was A Mistake

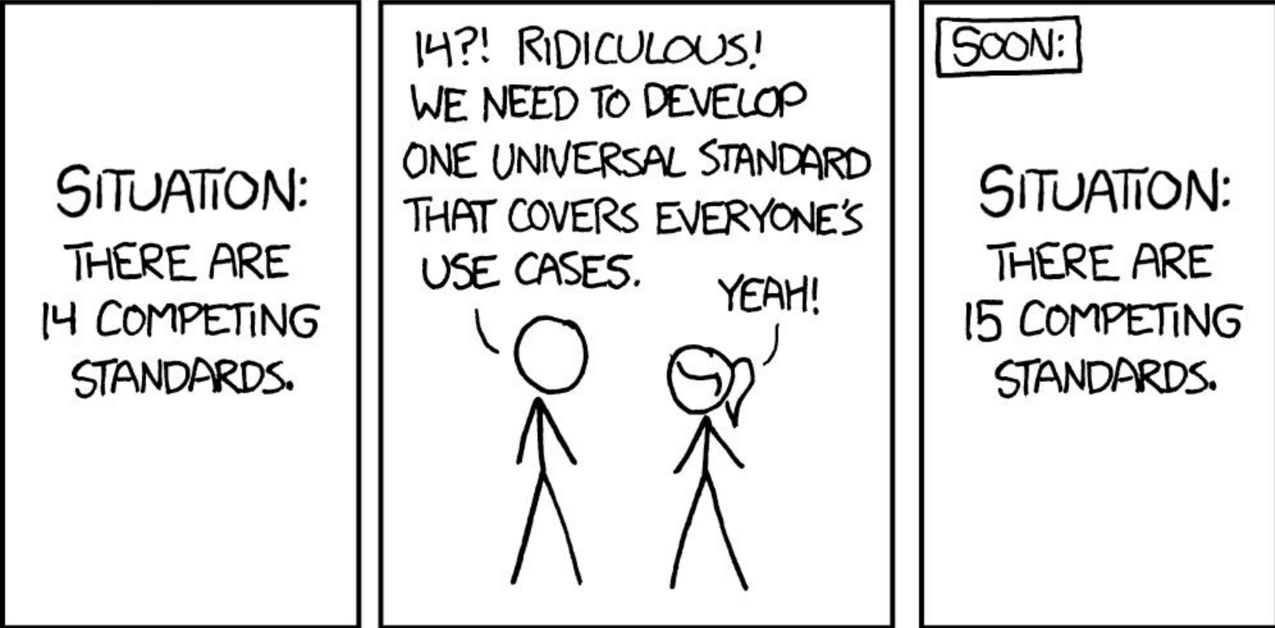
- Invariant: string always ends with a special character call the "null terminator"
- Violated if:
  - Buffer too small to contain string + terminator
  - null terminator overwritten with extra data
- O(n) to find length of string
- If invariant violated, trying to find length will cause a crash!
  - Very easy to violate these in user code with off-by-one errors
- Why we considered this at all: only 1 byte of overhead per string, data register can be re-used for loop guard

Explicit Length

- Invariant: length field is how many character units are valid
    - Usually, this field is private from user code
- Violated if:
    - Library operations don't update the length correctly
- O(1) to find length of string
- Drawback:
    - 4 or 8 bytes of overhead per string
    - For C compatibility, need to have null terminator anyways.

Character Encodings

Relevant XKCD



https://xkcd.com/927/

Relevant Wikipedia Article

Source encoding	Target encoding	Result	Occurrence
Hungarian example		ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP árvíztűrő tükörfúrógép	Characters in red are incorrect and do not match the top-left example.
CP 852	CP 437	╡RV╓ZTδRè TÜKÖRFΘRαGÉP árvízt√rï tükörfúrógép	This was very common in DOS-era when the text was encoded by the Central European CP 852 encoding; however, the operating system, a software or printer used the default CP 437 encoding. Please note that small-case letters are mainly correct, exception with ő (ï) and ű (√). Ü/ü is correct because CP 852 was made compatible with German. Nowadays occurs mainly on printed prescriptions and cheques.
CWI-2	CP 437	ÅRVìZTÿRº TÜKÖRFùRòGÉP árvíztûrô tükörfúrógép	The CWI-2 encoding was designed so that the text remains fairly well-readable even if the display or printer uses the default CP 437 encoding. This encoding was heavily used in the 1980s and early 1990s, but nowadays it is completely deprecated.
Windows-1250	Windows-1252	ÁRVÍZTÛRÕ TÜKÖRFÚRÓGÉP árvíztûrõ tükörfúrógép	The default Western Windows encoding is used instead of the Central-European one. Only ő-Ő (õ-Õ) and ű-Ű (û-Û) are wrong, but the text is completely readable. This is the most common error nowadays; due to ignorance, it occurs often on webpages or even in printed media.
CP 852	Windows-1250	µRVÖZTëRŠ TšK™RFéRŕG P rv˘ztür‹ t k"rfŁr˘g‚p	Central European Windows encoding is used instead of DOS encoding. The use of ű is correct.
Windows-1250	CP 852	┴RV═ZT█RŇ T▄KÍRF ┌RÉG╔P ßrvÝztűr§ tŘk÷rf˙r˘gÚp	Central European DOS encoding is used instead of Windows encoding. The use of ű is correct.
Quoted-printable	7-bit ASCII	=C1RV=CDZT=DBR=D5 T=DCK=D6RF=DAR=D3G=C9P =E1rv=EDzt=FBr=F5 t=FCk=F6rf=FAr=F3g=E9p	Mainly caused by wrongly configured mail servers but may occur in SMS messages on some cell-phones as well.
UTF-8	Windows-1252	ÃRVÃZTÅ°RÅ TÃœKÃ– RFÃšRÃ"GÃ‰P Ã¡rvÃztÅ±rÅ‘ tÃ¼kÃ¶rfÃºrÃ³gÃ©p	Mainly caused by wrongly configured web services or webmail clients, which were not tested for international usage (as the problem remains concealed for English texts). In this case the actual (often generated) content is in UTF-8; however, it is not configured in the HTML headers, so the rendering engine displays it with the default Western encoding.

https://en.wikipedia.org/wiki/Mojibake

But! A Standard Has Won!

- UTF-8
  - Used by over 97% of all websites
  - Default system encoding on Linux and MacOs
- Reasons:
  - Full compatibility with ASCII, which was historically very popular
  - 8-bit units => generally faster than wider encodings
  - Still flexible enough to represent more characters than we could realistically need

How UTF-8 Works

- Each possible character assigned a Unicode "code point"
  - "a" = U+0061
  - "🂫" = U+1F0AB
- Binary value of codepoint assigned to "x"s in following chart:

Code point <-> UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[nb 2]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

https://en.wikipedia.org/wiki/UTF-8

History And Competing Standards

- ASCII: **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
    - Developed from 1963-1968 for computerized telegraphs
    - 7-bit units
    - Sorting numerically => sorting alphabetically!
    - Room for control characters
    - "On March 11, 1968, U.S. President Lyndon B. Johnson mandated that all computers purchased by the United States Federal Government support ASCII"
- UTF-16
    - Unicode like UTF-8, but individual units are 2 bytes instead of 1
    - Not directly compatible with UTF-8
    - Pro: U+0000 to U+FFFF directly representable in 1 unit

C Supports All Standards

- … so long as they have a certain set of basic characters (English alphabet + numbers + some punctuation)
- and those basic characters take up exactly 1 byte
- and there's a NUL character with all zero bits to represent the end of strings
- Nothing else is enforced!
  - No ordering (besides digits), no non-negativity, definitely no encoding standard
- Reasonable to have these constraints, unreasonable that these are the only ones

C Literally Does Not Give A Darn

- String literals are compiled in, may not reflect encoding of system it runs on
- Converting between encodings is entirely optional
- Specifying the encoding you want is implementation-dependent
- C++ inherits all these flaws
    - std::string doesn't have any specified encoding!

Rust Does Give A Darn

- All strings are valid UTF-8
- Built-in functionality to convert to/from UTF-16 for interfacing with other languages

Rust's String Types

Borrowed String: `&str`

- Also called a "string slice"
- UTF-8 encoded
- Can refer to:
  - String literals in the program
  - Substrings of other strings

```
let x: &str = "Hello World";

let y: &str = &x[0..5];
```

Owned String: `String`

- UTF-8 encoded
- Allocated on the heap
- Dynamically resizable, like `Vec`

Rust's String Library

Many Ways Of Getting A `String`

```
let s1 = String::from("string literal");

let s2 = x.to_string(); // If x's type impls `ToString`,
automatically `impl`ed if `Display` is `impl`ed

let s3 = format!("concatenated {} and {}", s1, s2);

let s4 = String::from_utf8(vec![102, 111, 111]).unwrap();

let s5 = s1 + s2 + s3 + s4;

let s6 = s5.clone();
```

Standard Utilities: &str

- `len` (duh)
- `split` based on a pattern (arbitrary numbers of multi-character groups)
- `split_at` an index
- `to_uppercase` and `to_lowercase`
- Slice indexing thanks to the `Index` and `IndexMut` traits!
    - `let substr = s[5..10];`
- See https://doc.rust-lang.org/std/primitive.str.html for full list

Fancy Utilities: String

- `pop` last character
- `push_str` to append a new string
  - `+` is overloaded to this for String
- `truncate` a string to a new length
- Do everything an `&str` can do thanks to the `Deref` trait!
- See https://doc.rust-lang.org/std/string/struct.String.html for full list

The `FromStr` trait

```
pub trait FromStr {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

- Parsed struct cannot (safely) contain lifetimes: enforced by types!

Using the `FromStr` trait

```
// explicitly
let x1 = <i32 as FromStr>::from_str("42").unwrap();
// implicitly, through `str`'s `parse` method
let x2 = "42".parse::<i32>().unwrap();
```

- Pog! We just found a good use for the turbofish!

FFI String Types

For Compatibility With C:

- `CString`: Rust-owned string with no interior null bytes
  - Encoding not changed from UTF-8
  - Can use `as_bytes_with_nul` to get pointer to slice that ends with null terminator
  - Important! Cannot own a string that was created in C code
- `CStr`: C-owned string being borrowed in Rust
  - UTF-8 validation performed when converting to &str
  - Convertible to/from raw pointer
  - Aware of null terminator
- Requires realloc to add null terminator
- See https://doc.rust-lang.org/std/ffi/struct.CString.html and https://doc.rust-lang.org/std/ffi/struct.CStr.html for full info

Some `CStr` Methods

```
pub const fn as_ptr(&self) -> *const c_char

pub unsafe fn from_ptr<'a>(ptr: *const c_char) -> &'a
CStr

pub fn to_bytes_with_nul(&self) -> &[u8]

pub fn to_str(&self) -> Result<&str, Utf8Error>

// No .len() function!!
```

For Compatibility With The OS:

- `OSString`: Rust-owned string, no interior null bytes, platform encoding
  - Unix: assumed UTF-8
  - Windows: UTF-16 but encoded as WTF-8 for lossless conversion
- `OSStr`:
  - Borrowed version of the above
- Neither are aware of null terminators! Use ` CStr{ing}` if you need that
- See https://doc.rust-lang.org/std/ffi/struct.OsString.html and https://doc.rust-lang.org/std/ffi/struct.OsStr.html for full info

Some `OSStr` Methods

```
pub fn is_ascii(&self) -> bool

pub fn is_empty(&self) -> bool

pub fn len(&self) -> usize

pub fn to_str(&self) -> Option<&str>

// No .to_bytes() function!!
```

Paths

Paths Are Special Strings!

- Not all strings are valid paths though…
- So clearly we need a new type for this!
  - Wrapper around OSString to enforce invariants
- Owned version: `PathBuf` (like a `String`)
- Borrowed version: `Path` (like a `str`)

Some `Path` Methods

```
pub fn canonicalize(&self) -> Result<PathBuf>

pub fn exists(&self) -> bool

pub fn is_dir(&self) -> bool

pub fn is_file(&self) -> bool

pub fn join<P: AsRef<Path>>(&self, path: P) -> PathBuf
```

More "Fun" Traits

`Deref`

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

- Compiler will desugar `*v` into `Deref::deref(v)` when `v : &T`

`Deref`

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

- Compiler will desugar `*v` into `*Deref::deref(&v)` if appropriate
- `?Sized`: can be a "type with a size not known at compile time", like a slice type `[i32]`

Deref Coercion

- Type coercion: done when desired type is explicitly labeled, and casting would be lossless
    - `let x: i8 = 42; // 42 is an i32 literal, coerced to i8`
    - `fn foo(x: i8) {}; foo(42) // same for function arguments`
- Deref coercion is a subset: `&T` or `&mut T` can be coerced to `&U` if `T` implements `Deref<Target = U>`

How This Is Used In `std`

- `impl Deref<Target=str> for String`
- `impl Deref<Target=CStr> for CString`
- `impl Deref<Target=OSStr> for OSString`
- `impl Deref<Target=Path> for PathBuf`
- `impl Deref<Target=T> for &'_ T`
- TL;DR: If you want a function that takes in both `&str` and `&String`, you can just use `&str` and all your references will be automatically coerced!

How This Is Used In `std`

- impl Deref<Target=str> for String
- impl Deref<Target=CStr> for CString
- impl Deref<Target=OSStr> for OSString
- impl Deref<Target=Path> for PathBuf
- impl Deref<Target=T> for &'_ T
- TL;DR: If you want a function that takes in both `&str` and `&String`, you can just use `&str` and all your references will be automatically coerced!

`Borrow<Borrowed>`

```
pub trait Borrow<Borrowed> where Borrowed: ?Sized {
    fn borrow(&self) -> &Borrowed;
}
```

- Meant to be for wrapper types like `Box<T>` or `Rc<T>` where a reference to this acts *exactly* like a reference to the underlying type
- Requirements (not enforced by the compiler, by convention): `Eq`, `Hash`, and `Ord` implementations remain consistent

`AsRef<T>`

```
pub trait AsRef<T> where T: ?Sized {
    fn as_ref(&self) -> &T;
}
```

- If `T` implements `AsRef<U>`, this means you can get an `&U` for cheap from an `&T`
- No other guarantees!

Comparing These Traits

Trait	Deref	Borrow	AsRef
Convert by	compiler magic	.borrow()	.as_ref()
Informal requirements	only for "smart pointer"-like types	only when returned type has identical behavior wrt common traits	can be done fairly cheaply
Conversion can fail?	No	No	No
Mutable Version	DerefMut	BorrowMut	AsMut

Homework

All Assignments So Far Due Friday

- Have a good spring break!

Backup: Case Study In Encoding Errors

Not Just A Theoretical Issue

- https://github.com/tensorflow/tensorflow/issues/47022#issuecomment-939546658
- Tensorflow error: implicit whitespace splitting failed on certain unicode chars

```
>>> import tensorflow as tf
>>> tf.strings.split(["verità truth"], ' ')
<tf.RaggedTensor [[b'verit\xc3\xa0', b'truth']]>
>>> tf.strings.split(["verità truth"])
<tf.RaggedTensor [[b'verit\xc3']]>
```

This Algorithm Looks Correct, Yeah?

```
bool ConsumeNonWhitespace(StringPiece* s, StringPiece* val) {
  const char* p = s->data();
  const char* limit = p + s->size();
  while (p < limit) {
    const char c = *p;
    if (isspace(c)) break;
    p++;
  }
  const size_t n = p - s->data();
  if (n > 0) {
    *val = StringPiece(s->data(), n);
    s->remove_prefix(n);
    return true;
  } else {
    *val = StringPiece();
    return false;
  }
}
```

Well What If `isspace` Was Broken?

- This is exactly what happens on Windows-1252 codepage (default, English)
- Python: strings are UTF-8
  - `"verità" -> bytes([0x76, 0x65, 0x72, 0x69, 0x74, 0xc3, 0xa0])`
- In Windows-1252: 0xa0 is a non-breaking space, which does count as whitespace!
- There was a UTF-8 specific `isspace` somewhere else in the code, it just wasn't used lmao
- "Fixed" by making Windows use UTF-8 locale for everything

Backup: Why We Can't Index Individual Characters From &str

Wait We Can't?

- Nope, thanks UTF-8
- What would we even index on?
  - Individual bytes? Some aren't valid on their own
  - Individual "scalar values" (single codepoints)? Some modify other scalars! Know as combining characters, this is how Zalgo text works
  - Grapheme clusters? Wayyy too hard to put in a standard library, there will be crates for that
- Any default would exclude others, best to have first two as explicit functions: `bytes()` and `chars()`

But Slicing Still Works, Right?

- It's not guaranteed to!
- Slices are based on byte indexes
- Will panic if slice starts/ends in the middle of a scalar value
- All ASCII characters are 1 byte in UTF-8, so we don't notice this most of the time
- See https://doc.rust-lang.org/stable/book/ch08-02-strings.html for a more in-depth explanation