orphan:

Text Formatting in Swift

Author: Dave Abrahams
Author: Chris Lattner
Author: Dave Zarzycki
Date: 2013-08-12

Index

Abstract: We propose a system for creating textual representations of Swift objects. Our system unifies conversion to `String`, string interpolation, printing, and representation in the REPL and debugger.

Scope

Goals

- The REPL and LLDB ("debuggers") share formatting logic
- All types are "debug-printable" automatically
- Making a type "printable for humans" is super-easy
- `toString()`-ability is a consequence of printability.
- Customizing a type's printed representations is super-easy
- Format variations such as numeric radix are explicit and readable
- Large textual representations do not (necessarily) ever need to be stored in memory, e.g. if they're being streamed into a file or over a remote-debugging channel.

Non-Goals

- **Localization** issues such as pluralizing and argument presentation order are beyond the scope of this proposal.
- **Dynamic format strings** are beyond the scope of this proposal.
- **Matching the terseness of C**'s `printf` is a non-goal.

> **Rationale**
>
> Localization (including single-locale linguistic processing such as what's found in Clang's diagnostics subsystem) is the only major application we can think of for dynamically-constructed format strings, [3] and is certainly the most important consumer of that feature. Therefore, localization and dynamic format strings should be designed together, and *under this proposal* the only format strings are string literals containing interpolations ("`\(...)`"). Cocoa programmers can still use Cocoa localization APIs for localization jobs.
>
> In Swift, only the most common cases need to be very terse. Anything "fancy" can afford to be a bit more verbose. If

and when we address localization and design a full-featured dynamic string formatter, it may make sense to incorporate features of `printf` into the design.

CustomStringConvertible Types

`CustomStringConvertible` types can be used in string literal interpolations, printed with `print(x)`, and can be converted to `String` with `x.toString()`.

The simple extension story for beginners is as follows:

> "To make your type `CustomStringConvertible`, simply declare conformance to `CustomStringConvertible`:
>
> ```
> extension Person : CustomStringConvertible {}
> ```
>
> and it will have the same printed representation you see in the interpreter (REPL). To customize the representation, give your type a `func format()` that returns a `String`:
>
> ```
> extension Person : CustomStringConvertible {
>   func format() -> String {
>     return "\(lastName), \(firstName)"
>   }
> }
> ```

The formatting protocols described below allow more efficient and flexible formatting as a natural extension of this simple story.

Formatting Variants

`CustomStringConvertible` types with parameterized textual representations (e.g. number types) *additionally* support a `format(...)` method parameterized according to that type's axes of variability:

```
print(offset)
print(offset.format()) // equivalent to previous line
print(offset.format(radix: 16, width: 5, precision: 3))
```

Although `format(...)` is intended to provide the most general interface, specialized formatting interfaces are also possible:

```
print(offset.hex())
```

Design Details

Output Streams

The most fundamental part of this design is `TextOutputStream`, a thing into which we can stream text: [2]

```
protocol TextOutputStream {
  func append(_ text: String)
}
```

Every `String` can be used as an `TextOutputStream` directly:

```
extension String : TextOutputStream {
  func append(_ text: String)
}
```

Debug Printing

Via compiler magic, *everything* conforms to the `CustomDebugStringConvertible` protocol. To change the debug representation for a type, you don't need to declare conformance: simply give the type a `debugFormat()`:

```
/// A thing that can be printed in the REPL and the Debugger
protocol CustomDebugStringConvertible {
  typealias DebugRepresentation : TextOutputStreamable = String

  /// Produce a textual representation for the REPL and
  /// Debugger.
  func debugFormat() -> DebugRepresentation
}
```

Because `String` is a `TextOutputStreamable`, your implementation of `debugFormat` can just return a `String`. If want to write directly to the `TextOutputStream` for efficiency reasons, (e.g. if your representation is huge), you can return a custom `DebugRepresentation` type.

> **Guideline**

> Producing a representation that can be consumed by the REPL and LLDB to produce an equivalent object is strongly encouraged where possible! For example, `String.debugFormat()` produces a representation starting and ending with "`"`", where special characters are escaped, etc. A `struct Point { var x, y: Int }` might be represented as "`Point(x: 3, y: 5)`".

(Non-Debug) Printing

The `CustomStringConvertible` protocol provides a "pretty" textual representation that can be distinct from the debug format. For example, when `s` is a `String`, `s.format()` returns the string itself, without quoting.

Conformance to `CustomStringConvertible` is explicit, but if you want to use the `debugFormat()` results for your type's `format()`, all you need to do is declare conformance to `CustomStringConvertible`; there's nothing to implement:

```
/// A thing that can be print()ed and toString()ed.
protocol CustomStringConvertible : CustomDebugStringConvertible {
  typealias PrintRepresentation : TextOutputStreamable = DebugRepresentation

  /// produce a "pretty" textual representation.
  ///
  /// In general you can return a String here, but if you need more
  /// control, return a custom TextOutputStreamable type
  func format() -> PrintRepresentation {
    return debugFormat()
  }

  /// Simply convert to String
  ///
  /// You'll never want to reimplement this
  func toString() -> String {
    var result: String
    self.format().write(result)
    return result
  }
}
```

`TextOutputStreamable`

Because it's not always efficient to construct a `String` representation before writing an object to a stream, we provide a `TextOutputStreamable` protocol, for types that can write themselves into an `TextOutputStream`. Every `TextOutputStreamable` is also a `CustomStringConvertible`, naturally:

```
protocol TextOutputStreamable : CustomStringConvertible {
  func writeTo<T: TextOutputStream>(_ target: [inout] T)

  // You'll never want to reimplement this
  func format() -> PrintRepresentation {
    return this
  }
}
```

How `String` Fits In

`String`'s `debugFormat()` yields a `TextOutputStreamable` that adds surrounding quotes and escapes special characters:

```
extension String : CustomDebugStringConvertible {
  func debugFormat() -> EscapedStringRepresentation {
    return EscapedStringRepresentation(self)
  }
}

struct EscapedStringRepresentation : TextOutputStreamable {
  var _value: String

  func writeTo<T: TextOutputStream>(_ target: [inout] T) {
    target.append("\"")
    for c in _value {
      target.append(c.escape())
    }
    target.append("\"")
  }
}
```

Besides modeling `TextOutputStream`, `String` also conforms to `TextOutputStreamable`:

```
extension String : TextOutputStreamable {
  func writeTo<T: TextOutputStream>(_ target: [inout] T) {
    target.append(self) // Append yourself to the stream
  }
```

```
  func format() -> String {
    return this
  }
}
```

This conformance allows *most* formatting code to be written entirely in terms of `String`, simplifying usage. Types with other needs can expose lazy representations like `EscapedStringRepresentation` above.

Extended Formatting Example

The following code is a scaled-down version of the formatting code used for `Int`. It represents an example of how a relatively complicated `format(...)` might be written:

```
protocol CustomStringConvertibleInteger
  : ExpressibleByIntegerLiteral, Comparable, SignedNumber, CustomStringConvertible {
  func %(lhs: Self, rhs: Self) -> Self
  func /(lhs: Self, rhs: Self) -> Self
  constructor(x: Int)
  func toInt() -> Int

  func format(_ radix: Int = 10, fill: String = " ", width: Int = 0)
    -> RadixFormat<This> {

    return RadixFormat(this, radix: radix, fill: fill, width: width)
  }
}

struct RadixFormat<T: CustomStringConvertibleInteger> : TextOutputStreamable {
  var value: T, radix = 10, fill = " ", width = 0

  func writeTo<S: TextOutputStream>(_ target: [inout] S) {
    _writeSigned(value, &target)
  }

  // Write the given positive value to stream
  func _writePositive<T:CustomStringConvertibleInteger, S: TextOutputStream>(
    _ value: T, stream: [inout] S
  ) -> Int {
    if value == 0 { return 0 }
    var radix: T = T.fromInt(self.radix)
    var rest: T = value / radix
    var nDigits = _writePositive(rest, &stream)
    var digit = UInt32((value % radix).toInt())
    var baseCharOrd : UInt32 = digit <= 9 ? '0'.value : 'A'.value - 10
    stream.append(String(UnicodeScalar(baseCharOrd + digit)))
    return nDigits + 1
  }

  func _writeSigned<T:CustomStringConvertibleInteger, S: TextOutputStream>(
    _ value: T, target: [inout] S
  ) {
    var width = 0
    var result = ""

    if value == 0 {
      result = "0"
      ++width
    }
    else {
      var absVal = abs(value)
      if (value < 0) {
        target.append("-")
        ++width
      }
      width += _writePositive(absVal, &result)
    }

    while width < width {
      ++width
      target.append(fill)
    }
    target.append(result)
  }
}

extension Int : CustomStringConvertibleInteger {
  func toInt() -> Int { return this }
}
```

Possible Extensions (a.k.a. Complications)

We are not proposing these extensions. Since we have given them considerable thought, they are included here for completeness and to ensure our proposed design doesn't rule out important directions of evolution.

`TextOutputStream` Adapters

Most text transformations can be expressed as adapters over generic `TextOutputStream`s. For example, it's easy to imagine an upcasing adapter that transforms its input to upper case before writing it to an underlying stream:

```
struct UpperStream<UnderlyingStream:TextOutputStream> : TextOutputStream {
  func append(_ x: String) { base.append(x.toUpper()) }
  var base: UnderlyingStream
}
```

However, upcasing is a trivial example: many such transformations--such as `trim()` or regex replacement--are stateful, which implies some way of indicating "end of input" so that buffered state can be processed and written to the underlying stream:

```
struct TrimStream<UnderlyingStream:TextOutputStream> : TextOutputStream {
  func append(_ x: String) { ... }
  func close() { ... }
  var base: UnderlyingStream
  var bufferedWhitespace: String
}
```

This makes general `TextOutputStream` adapters more complicated to write and use than ordinary `TextOutputStream`s.

`TextOutputStreamable` Adapters

For every conceivable `TextOutputStream` adaptor there's a corresponding `TextOutputStreamable` adaptor. For example:

```
struct UpperStreamable<UnderlyingStreamable : TextOutputStreamable> {
  var base: UnderlyingStreamable

  func writeTo<T: TextOutputStream>(_ target: [inout] T) {
    var adaptedStream = UpperStream(target)
    self.base.writeTo(&adaptedStream)
    target = adaptedStream.base
  }
}
```

Then, we could extend `TextOutputStreamable` as follows:

```
extension TextOutputStreamable {
  typealias Upcased : TextOutputStreamable = UpperStreamable<This>
  func toUpper() -> UpperStreamable<This> {
    return Upcased(self)
  }
}
```

and, finally, we'd be able to write:

```
print(n.format(radix:16).toUpper())
```

The complexity of this back-and-forth adapter dance is daunting, and might well be better handled in the language once we have some formal model--such as coroutines--of inversion-of-control. We think it makes more sense to build the important transformations directly into `format()` methods, allowing, e.g.:

```
print(n.format(radix:16, case:.upper))
```

Possible Simplifications

One obvious simplification might be to fearlessly use `String` as the universal textual representation type, rather than having a separate `TextOutputStreamable` protocol that doesn't necessarily create a fully-stored representation. This approach would trade some efficiency for considerable design simplicity. It is reasonable to ask whether the efficiency cost would be significant in real cases, and the truth is that we don't have enough information to know. At least until we do, we opt not to trade away any CPU, memory, and power.

If we were willing to say that only `class`es can conform to `TextOutputStream`, we could eliminate the explicit `[inout]` where `TextOutputStream`s are passed around. Then, we'd simply need a `class StringStream` for creating `String` representations. It would also make `TextOutputStream` adapters a *bit* simpler to use because you'd never need to "write back" explicitly onto the target stream. However, stateful `TextOutputStream` adapters would still need a `close()` method, which makes a perfect place to return a copy of the underlying stream, which can then be "written back":

```
struct AdaptedStreamable<T : TextOutputStreamable> {
  ...
  func writeTo<Target: TextOutputStream>(_ target: [inout] Target) {
    // create the stream that transforms the representation
    var adaptedTarget = adapt(target, adapter);
    // write the Base object to the target stream
```

```
        base.writeTo(&adaptedTarget)
        // Flush the adapted stream and, in case Target is a value type,
        // write its new value
        target = adaptedTarget.close()
      }
      ...
    }
```

We think anyone writing such adapters can handle the need for explicit write-back, and the ability to use `String` as an `TextOutputStream` without additionally allocating a `StringStream` on the heap seems to tip the balance in favor of the current design.

---

[1] Whether `format(...)` is to be a real protocol or merely an ad-hoc convention is TBD. So far, there's no obvious use for a generic `format` with arguments that depend on the type being formatted, so an ad-hoc convention would be just fine.

[2] We don't support streaming individual code points directly because it's possible to create invalid sequences of code points. For any code point that, on its own, represents a valid `Character` (a.k.a. Unicode extended grapheme cluster), it is trivial and inexpensive to create a `String`. For more information on the relationship between `String` and `Character` see the (forthcoming, as of this writing) document *Swift Strings State of the Union*.

[3] In fact it's possible to imagine a workable system for localization that does away with dynamic format strings altogether, so that all format strings are fully statically-checked and some of the same formatting primitives can be used by localizers as by fully-privileged Swift programmers. This approach would involve compiling/JIT-ing localizations into dynamically-loaded modules. In any case, that will wait until we have native Swift dylibs.