orphan: We have a pretty good user model for C pointer interop now, but the language model still needs improvement. Building the user model on top of implicit conversions has a number of undesirable side effects. We end up with a mess of pointer types--the intended user-facing, one-word pointer types `UnsafeMutablePointer` and `OpaquePointer`, which expose a full pointer-ish API and are naturally ABI-compatible with C pointers; and the bridging pointer types, `ObjCMutablePointer`, `CMutablePointer`, `CConstPointer`, `CMutableVoidPointer`, and `CConstVoidPointer`, which have no real API yet but exist only to carry an owner reference and be implicitly convertible, and rely on compiler magic to be passed to C functions. Since we can do the magic pointer bridging only in limited places, we assault users writing method overrides and reading synthesized headers with both sets of pointer types in a confusing jumble.

The best solution to this is to burn the user model into the language, giving function applications special powers to provide the user model for pointers. We then provide only one set of plain pointer types, with special intrinsic behavior when used as function arguments.

The Pointer Types

In the standard library, we provide three pointer types:

- `UnsafePointer<T>`, corresponding to `T const *` in C and ARC,
- `UnsafeMutablePointer<T>`, corresponding to `T *` in C, and `T* __strong *` in ARC for class types, and
- `AutoreleasingUnsafeMutablePointer<T>` (for all `T: AnyObject`), corresponding to `T* __autoreleasing *` in ARC.

These types are all one word, have no ownership semantics, and share a common interface. `UnsafePointer` does not expose operations for storing to the referenced memory. `UnsafeMutablePointer` and `AutoreleasingUnsafeMutablePointer` differ in store behavior: `UnsafeMutablePointer` assumes that the pointed-to reference has ownership semantics, so `ptr.initialize(x)` consumes a reference to `x`, and `ptr.assign(x)` releases the originally stored value before storing the new value. `AutoreleasingUnsafeMutablePointer` assumes that the pointed-to reference does not have ownership semantics, so values are autoreleased before being stored by either initialize() or assign(), and no release is done on reassignment. Loading from any of the three kinds of pointer does a strong load, so there is no need for a separate `AutoreleasingUnsafePointer`.

Conversion behavior for pointer arguments

The user model for pointer arguments becomes an inherent capability of function applications. The rules are:

UnsafeMutablePointer<T>

When a function is declared as taking an `UnsafeMutablePointer<T>` argument, it can accept any of the following:

- `nil`, which is passed as a null pointer,
- an `UnsafeMutablePointer<T>` value, which is passed verbatim,
- an inout expression whose operand is a stored lvalue of type `T`, which is passed as the address of the lvalue, or
- an inout `Array<T>` value, which is passed as a pointer to the start of the array, and lifetime-extended for the duration of the callee.

As a special case, when a function is declared as taking an `UnsafeMutableRawPointer` argument, it can accept the same operands as `UnsafeMutablePointer<T>` for any type T.

So if you have a function declared:

```
func foo(_ x: UnsafeMutablePointer<Float>)
```

You can call it as any of:

```
var x: Float = 0.0
var p: UnsafeMutablePointer<Float> = nil
var a: [Float] = [1.0, 2.0, 3.0]
foo(nil)
foo(p)
foo(&x)
foo(&a)
```

And if you have a function declared:

```
func bar(_ x: UnsafeMutableRawPointer)
```

You can call it as any of:

```
var x: Float = 0.0, y: Int = 0
var p: UnsafeMutablePointer<Float> = nil, q: UnsafeMutablePointer<Int> = nil
var a: [Float] = [1.0, 2.0, 3.0], b: Int = [1, 2, 3]
bar(nil)
bar(p)
bar(q)
bar(&x)
```

```
bar(&y)
bar(&a)
bar(&b)
```

AutoreleasingUnsafeMutablePointer<T>

When a function is declared as taking an `AutoreleasingUnsafeMutablePointer<T>`, it can accept any of the following:

- nil, which is passed as a null pointer,
- an `AutoreleasingUnsafeMutablePointer<T>` value, which is passed verbatim, or
- an inout expression, whose operand is primitive-copied to a temporary nonowning buffer. The address of that buffer is passed to the callee, and on return, the value in the buffer is loaded, retained, and reassigned into the operand.

Note that the above list does not include arrays, since implicit autoreleasing-to-strong writeback of an entire array would probably not be desirable.

So if you have a function declared:

```
func bas(_ x: AutoreleasingUnsafeMutablePointer<NSBas?>)
```

You can call it as any of:

```
var x: NSBas?
var p: AutoreleasingUnsafeMutablePointer<NSBas?> = nil
bas(nil)
bas(p)
bas(&x)
```

UnsafePointer<T>

When a function is declared as taking an `UnsafeMutablePointer<T>` argument, it can accept any of the following:

- nil, which is passed as a null pointer,
- an `UnsafeMutablePointer<T>`, `UnsafePointer<T>`, or `AutoreleasingUnsafeMutablePointer<T>` value, which is converted to `UnsafePointer<T>` if necessary and passed verbatim,
- an inout expression whose operand is an lvalue of type `T`, which is passed as the address of (the potentially temporary writeback buffer of) the lvalue, or
- an `Array<T>` value, which is passed as a pointer to the start of the array, and lifetime-extended for the duration of the callee.

As a special case, when a function is declared as taking an `UnsafeRawPointer` argument, it can accept the same operands as `UnsafePointer<T>` for any type `T`. Pointers to certain integer types can furthermore interoperate with strings; see Strings below.

So if you have a function declared:

```
func zim(_ x: UnsafePointer<Float>)
```

You can call it as any of:

```
var x: Float = 0.0
var p: UnsafePointer<Float> = nil
zim(nil)
zim(p)
zim(&x)
zim([1.0, 2.0, 3.0])
```

And if you have a function declared:

```
func zang(_ x: UnsafeRawPointer)
```

You can call it as any of:

```
var x: Float = 0.0, y: Int = 0
var p: UnsafePointer<Float> = nil, q: UnsafePointer<Int> = nil
zang(nil)
zang(p)
zang(q)
zang(&x)
zang(&y)
let doubles = [1.0, 2.0, 3.0]
let ints = [1, 2, 3]
zang(doubles)
zang(ints)
```

A type checker limitation prevents array literals from being passed directly to `UnsafeRawPointer` arguments without type annotation. As a workaround, you can bind the array literal to a constant, as above, or specify the array type with `as`:

```
zang([1.0, 2.0, 3.0] as [Double])
zang([1, 2, 3] as [Int])
```

This limitation is tracked as <rdar://problem/17444930>.

Strings

Pointers to the following C integer and character types can interoperate with Swift `String` values and string literals:

- `CChar`, `CSignedChar`, and `CUnsignedChar`, which interoperate with `String` as a UTF-8 code unit array;
- (not implemented yet) `CShort`, `CUnsignedShort`, and `CChar16`, which interoperate with `String` as a UTF-16 code unit array; and
- (not implemented yet) `CInt`, `CUnsignedInt`, `CWideChar`, and `CChar32`, which interoperate with `String` as a UTF-32 code unit array.

A `UnsafePointer` parameter with any of the above element types may take a `String` value as an argument. The string is transcoded to a null-terminated buffer of the appropriate encoding, if necessary, and a pointer to the buffer is passed to the function. The callee may not mutate through the array, and the referenced memory is only guaranteed to live for the duration of the call.