Native code interoperability

Scala Native provides an interop layer that makes it easy to interact with foreign native code. This includes C and other languages that can expose APIs via C ABI (e.g. C++, D, Rust etc.)

All of the interop APIs discussed here are defined in `scala.scalanative.native` package. For brevity, we're going to refer to that namespace as just `native`.

Extern objects

Extern objects are simple wrapper objects that demarcate scopes where methods are treated as their native C ABI-friendly counterparts. They are roughly analogous to header files with top-level function declarations in C.

For example to call C's `malloc` one might declare it as following:

```
@native.extern
object libc {
  def malloc(size: native.CSize): native.Ptr[Byte] = native.extern
}
```

`native.extern` on the right hand side of the method definition signifies that the body of the method is defined elsewhere in a native library that is available on the library path (see Linking with native libraries.) Signature of the extern function must match the signature of the original C function (see Finding the right signature.)

Finding the right signature

To find a correct signature for a given C function one must provide an equivalent Scala type for each of the arguments:

C Type	Scala Type
void	Unit
bool	native.CBool
char, signed char	native.CChar
unsigned char	native.CUnsignedChar (1)
short	native.CShort
unsigned short	native.CUnsignedShort (1)
int	native.CInt
unsigned int	native.CUnsignedInt (1)
long	native.CLong
unsigned long	native.CUnsignedLong (1)
long long	native.CLongLong
unsigned long long	native.CUnsignedLongLong (1)
size_t	native.CSize
wchar_t	native.CWideChar
char16_t	native.CChar16
char32_t	native.CChar32
float	native.CFloat
double	native.CDouble
void*	native.Ptr[Byte] (2)
int*	native.Ptr[native.CInt] (2)
char*	native.CString (2) (3)
int (*)(int)	native.CFunctionPtr1[native.CInt, native.CInt] (2) (4)
struct { int x, y; }*	native.Ptr[native.CStruct2[native.CInt, native.CInt]] (2) (5)

C Type	Scala Type
struct { int x, y; }	Not supported

1. See Unsigned integer types.
2. See Pointer types.
3. See Byte strings.
4. See `Function pointers`_.
5. See Memory layout types.

Linking with native libraries

In C/C++ one has to typically pass an additional `-l mylib` flag to dynamically link with a library. In Scala Native one can annotate libraries to link with using `@native.link` annotation:

```
@native.link("mylib")
@native.extern
object mylib {
  ...
}
```

Whenever any of the members of `mylib` object are reachable, the Scala Native linker will automatically link with the corresponding native library.

Variadic functions

One can declare variadic functions like `printf` using `native.CVararg` auxiliary type:

```
@native.extern
object stdio {
  def printf(format: native.CString,
             args: native.CVararg*): native.CInt = native.extern
}
```

Pointer types

Scala Native provides a built-in equivalent of C's pointers via `native.Ptr[T]` data type. Under the hood pointers are implemented using unmanaged machine pointers.

Operations on pointers are closely related to their C counterparts and are compiled into equivalent machine code:

Operation	C syntax	Scala Syntax
Load value	`*ptr`	`!ptr`
Store value	`*ptr = value`	`!ptr = value`
Pointer to index	`ptr + i`, `&ptr[i]`	`ptr + i`
Load at index	`ptr[i]`	`ptr(i)`
Store at index	`ptr[i] = value`	`ptr(i) = value`
Pointer to field	`&ptr->name`	`ptr._N`
Load a field	`ptr->name`	`!ptr._N`
Store a field	`ptr->name = value`	`!ptr._N = value`

Where `N` is the index of the field `name` in the struct. See Memory layout types for details.

Memory management

Unlike standard Scala objects that are managed automatically by the underlying runtime system, one has to manage native pointers manually. The two standard ways to allocate memory in native code are:

v: latest

1. **Stack allocation.**

   Scala Native provides a built-in way to perform stack allocations of unmanaged memory using `native.stackalloc` function:

   ```
   val buffer = native.stackalloc[Byte](256)
   ```

   This code will allocate 256 bytes that are going to be available until the enclosing method returns. Number of elements to be allocated is optional and defaults to 1 otherwise.

   When using stack allocated memory one has to be careful not to capture this memory beyond the lifetime of the method. Dereferencing stack allocated memory after the method's execution has completed is undefined behaviour.

2. **Heap allocation.**

   Scala Native's library contains a bindings for a subset of the standard libc functionality. This includes the trio of `malloc`, `realloc` and `free` functions that are defined in `native.stdlib` extern object.

   Calling those will let you allocate memory using system's standard dynamic memory allocator. Apart from the system allocator one might also bind to pletheora of 3-rd party allocators such as jemalloc to serve the same purpose.

Undefined behavior

Similarly to their C counter-parts, behavior of operations that access memory is subject to undefined behaviour for following conditions:

1. Dereferencing null.
2. Out-of-bounds memory access.
3. Use-after-free.
4. Use-after-return.
5. Double-free, invalid free.

Memory layout types

Memory layout types are auxiliary types that let one specify memory layout of unmanaged memory. They are meant to be used purely in combination with native pointers and do not have a corresponding first-class values backing them.

- `native.Ptr[native.CStructN[T1, ..., TN]]`

  Pointer to a C struct with up to 22 fields. Type parameters are the types of corresponding fields. One may access fields of the struct using `_N` helper methods on a pointer value:

  ```
  val ptr = native.stackalloc[native.CStruct2[Int, Int]]
  !ptr._1 = 10
  !ptr._2 = 20
  println(s"first ${!ptr_.1}, second ${!ptr._2}")
  ```

  Here `_N` computes a derived pointer that corresponds to memory occupied by field number N.

- `native.Ptr[native.CArray[T, N]]`

  Pointer to a C array with statically-known length `N`. Length is encoded as a type-level natural number. Natural numbers are types that are composed of base naturals `Nat._0, ... Nat._9` and an additional `Nat.Digit` constructor. So for example number `1024` is going to be encoded as following:

  ```
  import scalanative.native._, Nat._

  type _1024 = Digit[_1, Digit[_0, Digit[_2, _4]]]
  ```

  Once you have a natural for the length, it can be used as an array length:

  ```
  val ptr = native.stackalloc[CArray[Byte, _1024]]
  ```

Addresses of the first twenty two elements are accessible via `_N` accessors. The rest are accessible via `ptr._1 + index`.

Byte strings

Scala Native supports byte strings via `c"..."` string interpolator that gets compiled down to pointers to statically-allocated zero-terminated strings (similarly to C):

```
import scalanative.native._

// CString is an alias to Ptr[CChar]
val msg: CString = c"Hello, world!"
stdio.printf(msg)
```

Additionally, we also expose two helper functions `native.toCString` and `native.fromCString` to convert between C-style and Java-style strings.

Unchecked casts

Quite often,C APIs expect user to perform unchecked casts to convert between different pointer types and/or pointers and integers values. We provide `obj.cast[T]` that's defined in `native.CCast` implicit class, for this use case. Unlike Scala's `asInstanceOf`, `cast` doesn't provide any safety guarantees.

Unsigned integer types

Scala Native provides support for four unsigned integer types:

1. `native.UByte`
2. `native.UShort`
3. `native.UInt`
4. `native.ULong`

They share the same primitive operations as signed integer types. Primitive operation between two integer values are supported only if they have the same signedness (they must both signed or both unsigned.)

Conversions between signed and unsigned integers must be done explicitly using `signed.toUByte`, `signed.toUShort`, `signed.toUInt`, `signed.toULong` and conversely `unsigned.toByte`, `unsigned.toShort`, `unsigned.toInt`, `unsigned.toLong`.

Continue to Libraries.