C Interface

Base.@ccall — Macro

```
@ccall library.function_name(argvalue1::argtype1, ...)::returntype
@ccall function_name(argvalue1::argtype1, ...)::returntype
@ccall $function_pointer(argvalue1::argtype1, ...)::returntype
```

Call a function in a C-exported shared library, specified by `library.function_name`, where `library` is a string constant or literal. The library may be omitted, in which case the `function_name` is resolved in the current process. Alternatively, `@ccall` may also be used to call a function pointer `$function_pointer`, such as one returned by `dlsym`.

Each `argvalue` to `@ccall` is converted to the corresponding `argtype`, by automatic insertion of calls to `unsafe_convert(argtype, cconvert(argtype, argvalue))`. (See also the documentation for `unsafe_convert` and `cconvert` for further details.) In most cases, this simply results in a call to `convert(argtype, argvalue)`.

Examples

```
@ccall strlen(s::Cstring)::Csize_t
```

This calls the C standard library function:

```
size_t strlen(char *)
```

with a Julia variable named `s`. See also `ccall`.

Varargs are supported with the following convention:

```
@ccall printf("%s = %d"::Cstring ; "foo"::Cstring, foo::Cint)::Cint
```

The semicolon is used to separate required arguments (of which there must be at least one) from variadic arguments.

Example using an external library:

```
# C signature of g_uri_escape_string:
# char *g_uri_escape_string(const char *unescaped, const char *reserved_chars_al

const glib = "libglib-2.0"
@ccall glib.g_uri_escape_string(my_uri::Cstring, ":/"::Cstring, true::Cint)::Cst
```

The string literal could also be used directly before the function name, if desired `"libglib-2.0".g_uri_escape_string(...`

`ccall` — Keyword

```
ccall((function_name, library), returntype, (argtype1, ...), argvalue1, ...)
ccall(function_name, returntype, (argtype1, ...), argvalue1, ...)
ccall(function_pointer, returntype, (argtype1, ...), argvalue1, ...)
```

Call a function in a C-exported shared library, specified by the tuple `(function_name, library)`, where each component is either a string or symbol. Instead of specifying a library, one can also use a `function_name` symbol or string, which is resolved in the current process. Alternatively, `ccall` may also be used to call a function pointer `function_pointer`, such as one returned by `dlsym`.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `argvalue` to the `ccall` will be converted to the corresponding `argtype`, by automatic insertion of calls to `unsafe_convert(argtype, cconvert(argtype, argvalue))`. (See also the documentation for `unsafe_convert` and `cconvert` for further details.) In most cases, this simply results in a call to `convert(argtype, argvalue)`.

`Core.Intrinsics.cglobal` — Function

```
cglobal((symbol, library) [, type=Cvoid])
```

Obtain a pointer to a global variable in a C-exported shared library, specified exactly as in `ccall`. Returns a `Ptr{Type}`, defaulting to `Ptr{Cvoid}` if no `Type` argument is supplied. The values can be read or written by `unsafe_load` or `unsafe_store!`, respectively.

`Base.@cfunction` — Macro

```
@cfunction(callable, ReturnType, (ArgumentTypes...,)) -> Ptr{Cvoid}
@cfunction($callable, ReturnType, (ArgumentTypes...,)) -> CFunction
```

Generate a C-callable function pointer from the Julia function `callable` for the given type signature. To pass the return value to a `ccall`, use the argument type `Ptr{Cvoid}` in the signature.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression (although it can include a splat expression). And that these arguments will be evaluated in global scope during compile-time (not deferred until runtime). Adding a '$' in front of the function argument changes this to instead create a runtime closure over the local variable `callable` (this is not supported on all architectures).

See manual section on ccall and cfunction usage.

Examples

```
julia> function foo(x::Int, y::Int)
           return x + y
       end

julia> @cfunction(foo, Int, (Int, Int))
Ptr{Cvoid} @0x000000001b82fcd0
```

`Base.CFunction` — Type

```
CFunction struct
```

Garbage-collection handle for the return value from `@cfunction` when the first argument is annotated with '$'. Like all `cfunction` handles, it should be passed to `ccall` as a `Ptr{Cvoid}`, and will be converted automatically at the call site to the appropriate type.

See `@cfunction`.

`Base.unsafe_convert` — Function

```
unsafe_convert(T, x)
```

Convert `x` to a C argument of type `T` where the input `x` must be the return value of `cconvert(T, ...)`.

In cases where `convert` would need to take a Julia object and turn it into a `Ptr`, this function should be used to define and perform that conversion.

Be careful to ensure that a Julia reference to `x` exists as long as the result of this function will be used. Accordingly, the argument `x` to this function should never be an expression, only a variable name or field reference. For example, `x=a.b.c` is acceptable, but `x=[a,b,c]` is not.

The `unsafe` prefix on this function indicates that using the result of this function after the `x` argument to this function is no longer accessible to the program may cause undefined behavior, including program corruption or segfaults, at any later time.

See also `cconvert`

`Base.cconvert` — Function

```
cconvert(T,x)
```

Convert `x` to a value to be passed to C code as type `T`, typically by calling `convert(T, x)`.

In cases where `x` cannot be safely converted to `T`, unlike `convert`, `cconvert` may return an object of a type different from `T`, which however is suitable for `unsafe_convert` to handle. The result of this function should be kept valid (for the GC) until the result of `unsafe_convert` is not needed anymore. This can be used to allocate memory that will be accessed by the `ccall`. If multiple objects need to be allocated, a tuple of the objects can be used as return value.

Neither `convert` nor `cconvert` should take a Julia object and turn it into a `Ptr`.

`Base.unsafe_load` — Function

```
unsafe_load(p::Ptr{T}, i::Integer=1)
```

Load a value of type `T` from the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1]`.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Incorrect usage may segfault your program or return garbage answers, in the same manner as C.

`Base.unsafe_store!` — Function

```
unsafe_store!(p::Ptr{T}, x, i::Integer=1)
```

Store a value of type `T` to the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1] = x`.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

`Base.unsafe_copyto!` — Method

```
unsafe_copyto!(dest::Ptr{T}, src::Ptr{T}, N)
```

Copy `N` elements from a source pointer to a destination, with no checking. The size of an element is determined by the type of the pointers.

The `unsafe` prefix on this function indicates that no validation is performed on the pointers `dest` and `src` to ensure that they are valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

`Base.unsafe_copyto!` — Method

```
unsafe_copyto!(dest::Array, do, src::Array, so, N)
```

Copy `N` elements from a source array to a destination, starting at offset `so` in the source and `do` in the destination (1-indexed).

The `unsafe` prefix on this function indicates that no validation is performed to ensure that N is inbounds on either array. Incorrect usage may corrupt or segfault your program, in the same manner as C.

`Base.copyto!` — Function

```
copyto!(dest::AbstractMatrix, src::UniformScaling)
```

Copies a `UniformScaling` onto a matrix.

Julia 1.1

In Julia 1.0 this method only supported a square destination matrix. Julia 1.1. added support for a rectangular matrix.

```
copyto!(dest, do, src, so, N)
```

Copy `N` elements from collection `src` starting at offset `so`, to array `dest` starting at offset `do`. Return `dest`.

```
copyto!(dest::AbstractArray, src) -> dest
```

Copy all elements from collection `src` to array `dest`, whose length must be greater than or equal to the length `n` of `src`. The first `n` elements of `dest` are overwritten, the other elements are left untouched.

Examples

```
julia> x = [1., 0., 3., 0., 5.];

julia> y = zeros(7);

julia> copyto!(y, x);

julia> y
7-element Array{Float64,1}:
 1.0
 0.0
```

```
3.0
0.0
5.0
0.0
0.0
```

```
copyto!(dest, Rdest::CartesianIndices, src, Rsrc::CartesianIndices) -> dest
```

Copy the block of `src` in the range of `Rsrc` to the block of `dest` in the range of `Rdest`. The sizes of the two regions must match.

`Base.pointer` — Function

```
pointer(array [, index])
```

Get the native address of an array or string, optionally at a given location `index`.

This function is "unsafe". Be careful to ensure that a Julia reference to `array` exists as long as this pointer will be used. The `GC.@preserve` macro should be used to protect the `array` argument from garbage collection within a given block of code.

Calling `Ref(array[, index])` is generally preferable to this function as it guarantees validity.

`Base.unsafe_wrap` — Method

```
unsafe_wrap(Array, pointer::Ptr{T}, dims; own = false)
```

Wrap a Julia `Array` object around the data at the address given by `pointer`, without making a copy. The pointer element type `T` determines the array element type. `dims` is either an integer (for a 1d array) or a tuple of the array dimensions. `own` optionally specifies whether Julia should take ownership of the memory, calling `free` on the pointer when the array is no longer referenced.

This function is labeled "unsafe" because it will crash if `pointer` is not a valid memory address to data of the requested length.

`Base.pointer_from_objref` — Function

```
pointer_from_objref(x)
```

Get the memory address of a Julia object as a `Ptr`. The existence of the resulting `Ptr` will not protect the object from garbage collection, so you must ensure that the object remains referenced for the whole time that the `Ptr` will be used.

This function may not be called on immutable objects, since they do not have stable memory addresses.

See also: `unsafe_pointer_to_objref`.

`Base.unsafe_pointer_to_objref` — Function

```
unsafe_pointer_to_objref(p::Ptr)
```

Convert a `Ptr` to an object reference. Assumes the pointer refers to a valid heap-allocated Julia object. If this is not the case, undefined behavior results, hence this function is considered "unsafe" and should be used with care.

See also: `pointer_from_objref`.

`Base.disable_sigint` — Function

```
disable_sigint(f::Function)
```

Disable Ctrl-C handler during execution of a function on the current task, for calling external code that may call julia code that is not interrupt safe. Intended to be called using `do` block syntax as follows:

```
disable_sigint() do
    # interrupt-unsafe code
    ...
end
```

This is not needed on worker threads (`Threads.threadid() != 1`) since the

`InterruptException` will only be delivered to the master thread. External functions that do not call julia code or julia runtime automatically disable sigint during their execution.

`Base.reenable_sigint` — Function

```
reenable_sigint(f::Function)
```

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of `disable_sigint`.

`Base.exit_on_sigint` — Function

```
exit_on_sigint(on::Bool)
```

Set `exit_on_sigint` flag of the julia runtime. If `false`, Ctrl-C (SIGINT) is capturable as `InterruptException` in `try` block. This is the default behavior in REPL, any code run via `-e` and `-E` and in Julia script run with `-i` option.

If `true`, `InterruptException` is not thrown by Ctrl-C. Running code upon such event requires `atexit`. This is the default behavior in Julia script run without `-i` option.

> **Julia 1.5**
>
> Function `exit_on_sigint` requires at least Julia 1.5.

`Base.systemerror` — Function

```
systemerror(sysfunc[, errno::Cint=Libc.errno()])
systemerror(sysfunc, iftrue::Bool)
```

Raises a `SystemError` for `errno` with the descriptive string `sysfunc` if `iftrue` is `true`

`Base.windowserror` — Function

```
windowserror(sysfunc[, code::UInt32=Libc.GetLastError()])
windowserror(sysfunc, iftrue::Bool)
```

Like `systemerror`, but for Windows API functions that use `GetLastError` to return an error code instead of setting `errno`.

`Core.Ptr` — Type

```
Ptr{T}
```

A memory address referring to data of type `T`. However, there is no guarantee that the memory is actually valid, or that it actually represents data of the specified type.

`Core.Ref` — Type

```
Ref{T}
```

An object that safely references data of type `T`. This type is guaranteed to point to valid, Julia-allocated memory of the correct type. The underlying data is protected from freeing by the garbage collector as long as the `Ref` itself is referenced.

In Julia, `Ref` objects are dereferenced (loaded or stored) with `[]`.

Creation of a `Ref` to a value `x` of type `T` is usually written `Ref(x)`. Additionally, for creating interior pointers to containers (such as Array or Ptr), it can be written `Ref(a, i)` for creating a reference to the `i`-th element of `a`.

When passed as a `ccall` argument (either as a `Ptr` or `Ref` type), a `Ref` object will be converted to a native pointer to the data it references.

There is no invalid (NULL) `Ref` in Julia, but a `C_NULL` instance of `Ptr` can be passed to a `ccall` Ref argument.

Use in broadcasting

`Ref` is sometimes used in broadcasting in order to treat the referenced values as a scalar:

```
julia> isa.(Ref([1,2,3]), [Array, Dict, Int])
3-element BitArray{1}:
 1
 0
 0
```

`Base.Cchar` — Type

```
Cchar
```

Equivalent to the native `char` c-type.

`Base.Cuchar` — Type

```
Cuchar
```

Equivalent to the native `unsigned char` c-type (`UInt8`).

`Base.Cshort` — Type

```
Cshort
```

Equivalent to the native `signed short` c-type (`Int16`).

`Base.Cstring` — Type

```
Cstring
```

A C-style string composed of the native character type `Cchar`s. `Cstring`s are NUL-terminated. For C-style strings composed of the native wide character type, see `Cwstring`. For more information about string interopability with C, see the manual.

`Base.Cushort` — Type

```
Cushort
```

Equivalent to the native `unsigned short` c-type (`UInt16`).

`Base.Cint` — Type

```
Cint
```

Equivalent to the native `signed int` c-type (`Int32`).

`Base.Cuint` — Type

```
Cuint
```

Equivalent to the native `unsigned int` c-type (`UInt32`).

`Base.Clong` — Type

```
Clong
```

Equivalent to the native `signed long` c-type.

`Base.Culong` — Type

```
Culong
```

Equivalent to the native `unsigned long` c-type.

`Base.Clonglong` — Type

```
Clonglong
```

Equivalent to the native `signed long long` c-type (`Int64`).

`Base.Culonglong` — Type

```
Culonglong
```

Equivalent to the native `unsigned long long` c-type (`UInt64`).

`Base.Cintmax_t` — Type

```
Cintmax_t
```

Equivalent to the native `intmax_t` c-type (`Int64`).

`Base.Cuintmax_t` — Type

```
Cuintmax_t
```

Equivalent to the native `uintmax_t` c-type (`UInt64`).

`Base.Csize_t` — Type

```
Csize_t
```

Equivalent to the native `size_t` c-type (`UInt`).

`Base.Cssize_t` — Type

```
Cssize_t
```

Equivalent to the native `ssize_t` c-type.

`Base.Cptrdiff_t` — Type

```
Cptrdiff_t
```

Equivalent to the native `ptrdiff_t` c-type (`Int`).

`Base.Cwchar_t` — Type

```
Cwchar_t
```

Equivalent to the native `wchar_t` c-type (`Int32`).

`Base.Cwstring` — Type

```
Cwstring
```

A C-style string composed of the native wide character type `Cwchar_t`s. `Cwstring`s are NUL-terminated. For C-style strings composed of the native character type, see `Cstring`. For more information about string interopability with C, see the manual.

`Base.Cfloat` — Type

```
Cfloat
```

Equivalent to the native `float` c-type (`Float32`).

`Base.Cdouble` — Type

```
Cdouble
```

Equivalent to the native `double` c-type (`Float64`).

LLVM Interface

`Core.Intrinsics.llvmcall` — Function

```
llvmcall(IR::String, ReturnType, (ArgumentType1, ...), ArgumentValue1, ...)
llvmcall((declarations::String, IR::String), ReturnType, (ArgumentType1, ...),
```

Call LLVM IR string in the first argument. Similar to an LLVM function `define` block, arguments are available as consecutive unnamed SSA variables (%0, %1, etc.).

The optional declarations string contains external functions declarations that are necessary for llvm to compile the IR string. Multiple declarations can be passed in by separating them with line breaks.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `ArgumentValue` to `llvmcall` will be converted to the corresponding `ArgumentType`, by automatic insertion of calls to `unsafe_convert(ArgumentType, cconvert(ArgumentType, ArgumentValue))`. (See also the documentation for `unsafe_convert` and `cconvert` for further details.) In most cases, this simply results in a call to `convert(ArgumentType, ArgumentValue)`.

See `test/llvmcall.jl` for usage examples.

« Iteration utilities C Standard Library »