Exposing a D Library to Python Through a C API

Ali Çehreli

DConf Online 2020

November 22

The speaker

With D since 2009

- Love at first sight: Created a Turkish D site[1], translated Andrei Alexandrescu's "The Case for D"[2] article to Turkish[3]

[1] http://ddili.org
[2] https://www.drdobbs.com/parallel/the-case-for-d/217801225
[3] http://ddili.org/makale/neden_d.html

The speaker

With D since 2009

- Love at first sight: Created a Turkish D site[1], translated Andrei Alexandrescu's "The Case for D"[2] article to Turkish[3]
- Known for the free book "Programming in D"[4]
  - "A happy accident"[5]
  - Recently available on Educative.io as an *interactive course*:
    - First part[6]
    - Second part[7]

[1] http://ddili.org
[2] https://www.drdobbs.com/parallel/the-case-for-d/217801225
[3] http://ddili.org/makale/neden_d.html
[4] http://ddili.org/ders/d.en/index.html
[5] https://dlang.org/blog/2016/06/29/programming-in-d-a-happy-accident/
[6] https://www.educative.io/courses/programming-in-d-ultimate-guide
[7] https://www.educative.io/collection/10370001/5620751206973440

The speaker (continued)

Currently at Mercedes-Benz Research and Development, North America

- Using D for ROS Bag File Manipulation for Autonomous Driving[1]

[1] https://dconf.org/2019/talks/cehreli.html

The speaker (continued)

Currently at Mercedes-Benz Research and Development, North America

- Using D for ROS Bag File Manipulation for Autonomous Driving[1]
- A project by Daimler and Bosch, a "happy place"

[1] https://dconf.org/2019/talks/cehreli.html

Use autowrap

```
import autowrap;
mixin(
    wrapDlang!(
        LibraryName("mylib"),
        Modules(
            Module("mymodule"),
            Module("myothermodule"),
        )
    )
);
```

Use `autowrap`

```
import autowrap;
mixin(
    wrapDlang!(
        LibraryName("mylib"),
        Modules(
            Module("mymodule"),
            Module("myothermodule"),
        )
    )
);
```

- Generates a Python extension as a shared library.
- Every D function marked as **`export`** in the modules **`mymodule`** and **`myothermodule`** are exposed as Python functions.
- Converts function names from **`camelCase`** to **`snake_case`**.
- Converts D exceptions to Python exceptions.
- Converts D structs and classes to Python classes.
- Python strings are passed to D functions from user code.

Use `autowrap` (continued)

Átila Neves's blog posts on **`autowrap`**[1]:

- The power of reflection[2]
- Want to call C from Python? Use D![3]

You are already there.

[1] https://github.com/symmetryinvestments/autowrap
[2] https://atilaoncode.blog/2020/01/22/the-power-of-reflection/
[3] https://atilaoncode.blog/2020/02/19/want-to-call-c-from-python-use-d/

Contents

1. Introduction
2. Providing D code as a library accessible from C
    - Symbols
    - Function interfaces
    - Error propagation
    - Lifetimes
    - Library interfaces
    - Initializing the D runtime
    - Example: Exposing a D range object to C
3. Calling from Python

Contents

1. Introduction
2. Providing D code as a library accessible from C
    - Symbols
    - Function interfaces
    - Error propagation
    - Lifetimes
    - Library interfaces
    - Initializing the D runtime
    - Example: Exposing a D range object to C
3. Calling from Python

Clicks,
not slides
▼
▼
▼

Compilation

Translating source code into object code (commonly, machine code.)

```
module deneme;

int add(int a, int b) {
  return a + b;
}
```

```
$ dmd -c deneme.d        ← Produces deneme.o
```

Compilation

Translating source code into object code (commonly, machine code.)

```
module deneme;

int add(int a, int b) {
  return a + b;
}
```

```
$ dmd -c deneme.d         ← Produces deneme.o
```

Using **obj2asm** that comes with **dmd**:

```
$ obj2asm deneme.o
[...]
_D6deneme3addFiiZi:                       ← Compiled deneme.add function
                push         RBP
                mov         RBP,RSP
                sub         RSP,8
                mov         -8[RBP],EDI
                mov         EAX,ESI
                add         EAX,-8[RBP]   ← Actual CPU instruction 'add'
[...]
```

Name mangling

Mangled function names are due to D's *overloading* feature.

```
int add(int a, int b) {
  return a + b;
}

double add(double a, double b) {
  return a + b;
}
```

Name mangling

Mangled function names are due to D's *overloading* feature.

```d
int add(int a, int b) {
  return a + b;
}

double add(double a, double b) {
  return a + b;
}
```

```
[...]
_D6deneme3addFiiZi:    ← Unique symbol for the 'int' overload
[...]
_D6deneme3addFddZd:    ← Unique symbol for the 'double' overload
[...]
```

See: D's Application Binary Interface (ABI) spec[1] for information on name mangling and more.

[1] https://dlang.org/spec/abi.html

Observing symbols

GNU Binutils **nm** program lists symbols in object files (including libraries and programs):

```
$ nm deneme.o
0000000000000000 R _D6deneme12__ModuleInfoZ      ← R: read only
0000000000000000 W _D6deneme3addFddZd            ← W: weak
0000000000000000 W _D6deneme3addFiiZi
                 U _d_dso_registry               ← U: undefined
                 U __start_minfo
                 U __stop_minfo
```

Observing symbols

GNU Binutils **nm** program lists symbols in object files (including libraries and programs):

```
$ nm deneme.o
0000000000000000 R _D6deneme12__ModuleInfoZ      ← R: read only
0000000000000000 W _D6deneme3addFddZd            ← W: weak
0000000000000000 W _D6deneme3addFiiZi
                 U _d_dso_registry               ← U: undefined
                 U __start_minfo
                 U __stop_minfo
```

__traits(getOverloads) and **.mangleof** of D, available at compile time:

```
static foreach (overload;   traits(getOverloads, deneme, "add")) {
  pragma(msg, overload.mangleof);
}
```

```
_D6deneme3addFiiZi
_D6deneme3addFddZd
```

Linker

Combines object files to make an executable file.

Linker

Combines object files to make an executable file.

Assume **`main.d`** alongside the earlier **`deneme.d`**:

```
import deneme;

void main() {
  add(1, 2);    // Actual call is _D6deneme3addFiiZi(1, 2)
}
```

Linker

Combines object files to make an executable file.

Assume **main.d** alongside the earlier **deneme.d**:

```
import deneme;

void main() {
  add(1, 2);    // Actual call is _D6deneme3addFiiZi(1, 2)
}
```

Separate compilation:

```
$ dmd -c deneme.d    ← Defines _D6deneme3addFiiZi
$ dmd -c main.d      ← Calls   _D6deneme3addFiiZi
```

Linker

Combines object files to make an executable file.

Assume **main.d** alongside the earlier **deneme.d**:

```
import deneme;

void main() {
  add(1, 2);    // Actual call is _D6deneme3addFiiZi(1, 2)
}
```

Separate compilation:

```
$ dmd -c deneme.d    ← Defines _D6deneme3addFiiZi
$ dmd -c main.d      ← Calls   _D6deneme3addFiiZi
```

Linker is almost never seen because it is called by **dmd** automatically:

```
$ dmd main.o deneme.o -ofmy_program
```

Language differences

Languages and compilers are free to choose name mangling schemes:

Language	**int add(int, int)**	**double add(double, double)**
D with **dmd**	_D6deneme3addFiiZi	_D6deneme3addFddZd
C++ with **g++**	_Z3addii	_Z3adddd
C with **gcc**	add	sorry, no overloading

Language differences

Languages and compilers are free to choose name mangling schemes:

Language	`int add(int, int)`	`double add(double, double)`
D with **dmd**	`_D6deneme3addFiiZi`	`_D6deneme3addFddZd`
C++ with **g++**	`_Z3addii`	`_Z3adddd`
C with **gcc**	`add`	sorry, no overloading

- Historically, the common language is C.
- The lack of overloading in C requires *manual name mangling*.

extern(C)

```
extern(C) int add_int(int a, int b) {
  return a + b;
}

extern(C) double add_double(double a, double b) {
  return a + b;
}
```

extern(C)

```
extern(C) int add_int(int a, int b) {
  return a + b;
}

extern(C) double add_double(double a, double b) {
  return a + b;
}
```

```
add_int
add_double
```

`extern(C)`

```
extern(C) int add_int(int a, int b) {
  return a + b;
}

extern(C) double add_double(double a, double b) {
  return a + b;
}
```

```
add_int
add_double
```

No limitation for **`extern(C)`** function bodies; they can be as D as needed:

```
extern(C) int foo(int * result) {
  *result = 5.iota.sum;    // ← D range algorithms
  return 0;
}
```

*Note: The **`extern(C)`** "linkage attribute" involves more than just name mangling; see Linkage Attribute spec[1] for more information.*

*Note: There is also **`extern(C++)`**.*

[1] https://dlang.org/spec/attribute.html#linkage

`extern(C)` function interfaces

Fundamental D types that have C counterparts:

- `int`, `double`, etc. (Careful: D types have exact widths, C types do not; use names like `int32_t` from `stdint.h`.)

`extern(C)` function interfaces

Fundamental D types that have C counterparts:

- `int`, `double`, etc. (Careful: D types have exact widths, C types do not; use names like `int32_t` from `stdint.h`.)
- `struct`

`extern(C)` function interfaces

Fundamental D types that have C counterparts:

- `int`, `double`, etc. (Careful: D types have exact widths, C types do not; use names like `int32_t` from `stdint.h`.)
- `struct`
- `function` (same as C function pointer)

`extern(C)` function interfaces

Fundamental D types that have C counterparts:

- **`int`**, **`double`**, etc. (Careful: D types have exact widths, C types do not; use names like **`int32_t`** from **`stdint.h`**.)
- **`struct`**
- **`function`** (same as C function pointer)
- Arrays as a pair of length (**`size_t`**) and *pointer to first element*

`extern(C)` function interfaces

Fundamental D types that have C counterparts:

- **`int`**, **`double`**, etc. (Careful: D types have exact widths, C types do not; use names like **`int32_t`** from **`stdint.h`**.)
- **`struct`**
- **`function`** (same as C function pointer)
- Arrays as a pair of length (**`size_t`**) and *pointer to first element*
- Strings either as arrays or zero-terminated strings

`extern(C)` function interfaces

Fundamental D types that have C counterparts:

- **`int`**, **`double`**, etc. (Careful: D types have exact widths, C types do not; use names like **`int32_t`** from **`stdint.h`**.)
- **`struct`**
- **`function`** (same as C function pointer)
- Arrays as a pair of length (**`size_t`**) and *pointer to first element*
- Strings either as arrays or zero-terminated strings

Not supported:

- Associative arrays, **`class`**, **`delegate`**

See: Data Type Compatibility[1] for more information.

[1] https://dlang.org/spec/interfaceToC.html#data_type_compat

Example

D library function:

```
extern(C) void D_func(size_t length,        // 1a) Array length
                      int * ptr,            // 1b) Array pointer
                      const char * strz,    // 2)  String
                      double * result) {    // 3)  "Out" parameter
```

Example

D library function:

```
extern(C) void D_func(size_t length,        // 1a) Array length
                      int * ptr,            // 1b) Array pointer
                      const char * strz,    // 2)  String
                      double * result) {    // 3)  "Out" parameter
```

```
  auto arr = ptr[0..length];    // 1) Slice from pointer+length (no copy, minimal cost)
```

Example

D library function:

```
extern(C) void D_func(size_t length,        // 1a) Array length
                      int * ptr,            // 1b) Array pointer
                      const char * strz,    // 2)  String
                      double * result) {    // 3)  "Out" parameter
```

```
  auto arr = ptr[0..length];    // 1) Slice from pointer+length (no copy, minimal cost)
```

```
  auto str = strz.fromStringz;  // 2) string from zero-terminated string
                                //    (does not copy but counts characters until '\0')
```

Example

D library function:

```d
extern(C) void D_func(size_t length,        // 1a) Array length
                      int * ptr,            // 1b) Array pointer
                      const char * strz,    // 2)  String
                      double * result) {    // 3)  "Out" parameter
```

```d
  auto arr = ptr[0..length];    // 1) Slice from pointer+length (no copy, minimal cost)
```

```d
  auto str = strz.fromStringz;  // 2) string from zero-terminated string
                                //    (does not copy but counts characters until '\0')
```

```d
  *result = 2.5;                // 3)
}
```

Example

D library function:

```
extern(C) void D_func(size_t length,        // 1a) Array length
                      int * ptr,            // 1b) Array pointer
                      const char * strz,    // 2)  String
                      double * result) {    // 3)  "Out" parameter
```

```
  auto arr = ptr[0..length];    // 1) Slice from pointer+length (no copy, minimal cost)
```

```
  auto str = strz.fromStringz;  // 2) string from zero-terminated string
                                //    (does not copy but counts characters until '\0')
```

```
  *result = 2.5;                // 3)
}
```

The C header file of this D library:

```
// mylibrary/mylibrary.h
#pragma once

#include <stddef.h> // For size_t
#include <stdint.h> // For int32_t

void D_func(size_t length, int32_t * ptr, const char * strz, double * result);
```

Example (continued)

C code, using this D library:

```c
#include <mylibrary/mylibrary.h>  // D library API

void C_func() {
  int32_t arr[] = { 0, 1, 2 };
  size_t length = 3;     // or ARRAY_SIZE(arr) if available

  double result = 0;

  D_func(length,         // 1a) Array length
         arr,            // 1b) Array pointer
         "hello",        // 2)  String
         &result);       // 3)  "Out" parameter
}
```

Error propagation

D's exception hierarchy:

```
            Throwable
           ↗         ↖
   Exception         Error
    ↗    ↖           ↗    ↖
  ...    ...       ...    ...
(recoverable)    (irrecoverable)
```

Error propagation

D's exception hierarchy:

```
            Throwable
           ↗         ↖
   Exception         Error
    ↗    ↖           ↗    ↖
  ...    ...       ...    ...
(recoverable)    (irrecoverable)
```

Thanks to exceptions, D functions normally *return* results:

```d
MyResult foo() {
  // ...
  enforce(cond, format!"Invalid: %s"(a));  // Throws Exception
  // ...
  assert(x == 42, "Invalid x!");           // Throws Error
  // ...
  return MyResult(42);
}
```

Error propagation (continued)

C does not have exceptions:

- Return value is reserved for the error code.
- So, functions must *return* their results as out parameters.

```
// C code:
int foo(MyResult * result) {
  // ...
  if (!cond) {
    return 1;                   // Non-zero: failure
  }
  *result = MyResult(42);
  return 0;                     // Zero: success
}
```

Error propagation (continued)

D exceptions must be translated to error codes.

Error propagation (continued)

D exceptions must be translated to error codes.

```
extern(C) int foo(MyResult * result) {
  try {
    // ...
    *result = MyResult(42);
    return 0;
```

Error propagation (continued)

D exceptions must be translated to error codes.

```
extern(C) int foo(MyResult * result) {
  try {
    // ...
    *result = MyResult(42);
    return 0;
```

```
  } catch (Exception exc) {
    stderr.writefln!"ERROR: %s"(exc.msg);  // Does stderr even exist? (Next
    return 1;                              // slide will return the message.)
```

Error propagation (continued)

D exceptions must be translated to error codes.

```
extern(C) int foo(MyResult * result) {
  try {
    // ...
    *result = MyResult(42);
    return 0;
```

```
  } catch (Exception exc) {
    stderr.writefln!"ERROR: %s"(exc.msg);  // Does stderr even exist? (Next
    return 1;                              // slide will return the message.)
```

```
  } catch (Error err) {
    stderr.writefln!"ERROR: %s"(err);

    // a)  abort();     Are we allowed to kill the caller's program?
    // b)  return 2;    Is this good and responsible enough?
  }
}
```

Perhaps the library's **`Error`** behavior should be configurable.

Status return type

Better than just **int** code:

```
struct Status {
  int code;
  const(char) * errMsg;
}

extern(C) Status foo(MyResult * result) {
  // ...
}
```

Status return type

Better than just **int** code:

```
struct Status {
  int code;
  const(char) * errMsg;
}

extern(C) Status foo(MyResult * result) {
  // ...
}
```

C definition is almost identical:

```
// mylibrary/mylibrary.h

#include <stdint.h> // For int32_t

typedef struct {
  int32_t code;
  const char * errMsg;
} Status;
```

`nothrow`

- Guarantees that the function does not emit any exception derived from **`Exception`**
- May still emit exceptions derived from **`Error`**

nothrow

- Guarantees that the function does not emit any exception derived from **Exception**
- May still emit exceptions derived from **Error**

```
nothrow extern(C) Status foo(MyResult * result) {
  // ...
}
```

tried function template

```
nothrow
Status tried(Func)(Func func,
                   string functionName = __FUNCTION__) {
  try {
    func();
    return Status(0, "Success");
```

tried function template

```
nothrow
Status tried(Func)(Func func,
                   string functionName = __FUNCTION__) {
  try {
    func();
    return Status(0, "Success");
```

```
  } catch (Exception exc) {
    return Status(1, exc.msg.toStringz);
```

tried function template

```
nothrow
Status tried(Func)(Func func,
                   string functionName = __FUNCTION__) {
  try {
    func();
    return Status(0, "Success");
```

```
  } catch (Exception exc) {
    return Status(1, exc.msg.toStringz);
```

```
  } catch (Error err) {
    // Non-throwing printing without zero-terminated strings:
    import core.stdc.stdio;
    import core.stdc.stdlib;
    fprintf(stderr, "\n%.*s(%zu): Failed to execute %.*s: %.*s\n",
            cast(int)err.file.length, err.file.ptr,
            err.line,
            cast(int)functionName.length, functionName.ptr,
            cast(int)err.msg.length, err.msg.ptr);
    abort();
  }

  assert(false);
}
```

tried function template (continued)

With the **tried** template, all library functions can be lambdas passed to **tried**:

```
nothrow extern(C) Status foo(MyResult * result) {
  return tried({
    // ...
    *result = MyResult(42);
  });
}
```

Argument lifetimes

```
nothrow extern(C)
Status foo(const(char) * name,        // 1)  string

           size_t length,             // 2a) Array of strings
           const(char) ** strings) {  // 2b)
  // ...
}
```

Argument lifetimes

```
nothrow extern(C)
Status foo(const(char) * name,        // 1)  string

           size_t length,             // 2a) Array of strings
           const(char) ** strings) {  // 2b)
  // ...
}
```

`fromStringz` means "make D string from zero terminated string". It is fine for *immediate use*.

```
  writefln!"name: %s"(name.fromStringz);  // 1) no copy
```

Argument lifetimes

```
nothrow extern(C)
Status foo(const(char) * name,        // 1)  string

           size_t length,             // 2a) Array of strings
           const(char) ** strings) {  // 2b)
  // ...
}
```

`fromStringz` means "make D string from zero terminated string". It is fine for *immediate use*.

```
  writefln!"name: %s"(name.fromStringz);  // 1) no copy
```

A range of D strings from C array of C strings:

```
  writefln!"array: %-(\n  %s%)"(strings[0..length].map!(s => s.fromStringz));
                                  //  2) no copy                 no copy
```

Argument lifetimes (continued)

`fromStringz` is NOT for *storing* for later use.

```
File file;
string[] arr;

nothrow extern(C)
Status foo(const(char) * name,
           size_t length,
           const(char) ** strings) {
  // ...
  file = File(name.fromStringz.idup);    // Copies
  // ...
}
```

Argument lifetimes (continued)

`fromStringz` is NOT for *storing* for later use.

```
File file;
string[] arr;

nothrow extern(C)
Status foo(const(char) * name,
           size_t length,
           const(char) ** strings) {
  // ...
  file = File(name.fromStringz.idup);    // Copies
  // ...
}
```

A D array of D strings from C array of C strings:

```
  arr = strings[0..length].map!(s => s.fromStringz.idup).array;
                                           //       ↑      ↑
                                           //   copies    allocates
```

Note: As an optimization exercise, all D strings as well as the D array can be inside a single memory block.

D object lifetimes

`toStringz` means "make zero terminated string from D string". It is fine for *immediate use* on the C side.

```
nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz; // Allocates from the GC
  // ...
}
```

D object lifetimes

toStringz means "make zero terminated string from D string". It is fine for *immediate use* on the C side.

```
nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz; // Allocates from the GC
  // ...
}
```

The GC will release unreferenced objects.

- Must document that the caller should make a copy if it needs the content for later use.

D object lifetimes (continued)

GC resources are not safe to *store* on the C side as-is.

D object lifetimes (continued)

GC resources are not safe to *store* on the C side as-is.

Options:

a) Store on the D side as well:

```
const(char) * n;

nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz;
  n = *name;    // Will be alive as long as 'n' keeps the reference.
  // ...
}
```

D object lifetimes (continued)

GC resources are not safe to *store* on the C side as-is.

Options:

a) Store on the D side as well:

```
const(char) * n;

nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz;
  n = *name;    // Will be alive as long as 'n' keeps the reference.
  // ...
}
```

b) Be explicit about it:

```
nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz;
  GC.addRoot(*name);    // Mark as "in use". (Call GC.removeRoot() later.)
  // ...
}
```

Is toStringz a pessimization?

```
nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz;  // Allocates and copies
  // ...
}
```

Is toStringz a pessimization?

```
nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz;  // Allocates and copies
  // ...
}
```

```
  auto s = makeString(42);
  s ~= '\0';                          // Sometimes no allocation
  *name = s.ptr;
```

Is toStringz a pessimization?

```
nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz;  // Allocates and copies
  // ...
}
```

```
  auto s = makeString(42);
  s ~= '\0';                            // Sometimes no allocation
  *name = s.ptr;
```

```
  *name = format!"file%s.txt\0"(42).ptr;    // Likely no allocation
```

Is toStringz a pessimization?

```
nothrow extern(C) Status bar(const(char) ** name) {
  // ...
  *name = makeString(42).toStringz;  // Allocates and copies
  // ...
}
```

```
  auto s = makeString(42);
  s ~= '\0';                          // Sometimes no allocation
  *name = s.ptr;
```

```
  *name = format!"file%s.txt\0"(42).ptr;    // Likely no allocation
```

```
  *name = "hello";    // String literals are already zero-terminated.
                      // Also note, no .ptr is necessary.
```

C library interfaces

Similar to object oriented design, library functionality usually involves

- Some state
- Functions that work with that state

C library interfaces

Similar to object oriented design, library functionality usually involves

- Some state
- Functions that work with that state

Related, a library interface involves

1. An **opaque handle** that represents that state
2. Initialization of that state (**constructor**)
3. Deinitialization of that state (**destructor**)
4. **Functions** that work with that state

C library interfaces

Similar to object oriented design, library functionality usually involves

- Some state
- Functions that work with that state

Related, a library interface involves

1. An **opaque handle** that represents that state
2. Initialization of that state (**constructor**)
3. Deinitialization of that state (**destructor**)
4. **Functions** that work with that state

If the program is *not* linked with a D compiler (e.g. `dmd`):

5. Initialization of the D runtime
6. Deinitialization of the D runtime

Library example

Let's expose the following D functionality as a C library:

```d
auto lineRange(string fileName = null) {
  enforce(!fileName.empty, "Empty file name.");

  return File(fileName)
         .byLine
         .map!strip
         .filter!(line => !line.empty)
         .filter!(line => !line.startsWith('#'));    // ← The range object
}
```

Library example

Let's expose the following D functionality as a C library:

```d
auto lineRange(string fileName = null) {
  enforce(!fileName.empty, "Empty file name.");

  return File(fileName)
         .byLine
         .map!strip
         .filter!(line => !line.empty)
         .filter!(line => !line.startsWith('#'));    // ← The range object
}
```

We will provide 5 functions to C, the equivalents of the following:

1. A constructor
2. A destructor
3. **`empty`**
4. **`front`**
5. **`popFront`**

`struct` wrapper

We might want to use the range object as our *opaque handle*. However:

- Cannot make a range object opaque as-is.
- Cannot `new` such objects that are usually by-value.
- In general, there is more state that goes along with this object.
- In general, there is some additional behavior e.g. data translation.

`struct wrapper` (continued)

So, we will wrap it in a D struct:

```
struct LineRange {
  alias LR = typeof(lineRange());    // ← Unmentionable type
  LR lr;                             // ← The wrapped object

  // ...
}
```

struct wrapper (continued)

So, we will wrap it in a D struct:

```
struct LineRange {
  alias LR = typeof(lineRange());    // ← Unmentionable type
  LR lr;                             // ← The wrapped object

  // ...
}
```

C header uses opaque type:

```
// mylibrary/mylibrary.h

typedef void* LineRange;    // NOTE: Could be simply 'void'
```

struct wrapper (continued)

```
struct LineRange {
  alias LR = typeof(lineRange());
  LR lr;

  this(LR lr) {
    this.lr = lr;
    prime();
  }

  void prime() {
    if (lr.empty) {
      this.front = null;

    } else {
      this.front = lr.front.toStringz;
    }
  }

  // The InputRange functionality follows.
```

struct wrapper (continued)

```
struct LineRange {
  alias LR = typeof(lineRange());
  LR lr;

  this(LR lr) {
    this.lr = lr;
    prime();
  }

  void prime() {
    if (lr.empty) {
      this.front = null;

    } else {
      this.front = lr.front.toStringz;
    }
  }

  // The InputRange functionality follows.
```

```
  auto empty() {
    return lr.empty;
  }

  const(char) * front;

  void popFront() {
    lr.popFront();
    prime();
  }
}
```

1/5 - Constructor

D code:

```
nothrow extern(C)
Status LineRange_ctor(LineRange ** lr,
                      const(char*) fileName) {
  return tried({                               ← 1
    enforce(lr, "NULL LineRange pointer.");    ← 2
    enforce(fileName, "NULL file name.");

    *lr = new LineRange(lineRange(fileName.fromStringz.idup));
    GC.addRoot(*lr);                  ↑ 3
  });     ↑ 4
}
```

1/5 - Constructor

D code:

```
nothrow extern(C)
Status LineRange_ctor(LineRange ** lr,
                      const(char*) fileName) {
  return tried({                               ← 1
    enforce(lr, "NULL LineRange pointer.");    ← 2
    enforce(fileName, "NULL file name.");

    *lr = new LineRange(lineRange(fileName.fromStringz.idup));
    GC.addRoot(*lr);                  ↑ 3
  });     ↑ 4
}
```

C header:

```
// mylibrary/mylibrary.h
Status LineRange_ctor(LineRange * range, const char * fileName);
```

1/5 - Constructor

D code:

```
nothrow extern(C)
Status LineRange_ctor(LineRange ** lr,
                      const(char*) fileName) {
  return tried({                               ← 1
    enforce(lr, "NULL LineRange pointer.");    ← 2
    enforce(fileName, "NULL file name.");

    *lr = new LineRange(lineRange(fileName.fromStringz.idup));
    GC.addRoot(*lr);                  ↑ 3
  });     ↑ 4
}
```

C header:

```
// mylibrary/mylibrary.h
Status LineRange_ctor(LineRange * range, const char * fileName);
```

C user example:

```
  LineRange lr = NULL;
  status = LineRange_ctor(&lr, "myfile.txt");
```

2/5 - Destructor

D code:

```
nothrow extern(C) Status LineRange_dtor(LineRange * lr) {
  return tried({
    if (lr) {
      destroy(*lr);  // (destroy() is usually unnecessary.)
                     // NOTE: destroy(lr) would be wrong.
      GC.removeRoot(lr);
    }
  });
}
```

2/5 - Destructor

D code:

```d
nothrow extern(C) Status LineRange_dtor(LineRange * lr) {
  return tried({
    if (lr) {
      destroy(*lr);  // (destroy() is usually unnecessary.)
                     // NOTE: destroy(lr) would be wrong.
      GC.removeRoot(lr);
    }
  });
}
```

C header:

```c
// mylibrary/mylibrary.h

Status LineRange_dtor(LineRange range);
```

2/5 - Destructor

D code:

```d
nothrow extern(C) Status LineRange_dtor(LineRange * lr) {
  return tried({
    if (lr) {
      destroy(*lr);  // (destroy() is usually unnecessary.)
                     // NOTE: destroy(lr) would be wrong.
      GC.removeRoot(lr);
    }
  });
}
```

C header:

```c
// mylibrary/mylibrary.h

Status LineRange_dtor(LineRange range);
```

C user example:

```c
  status = LineRange_dtor(lr);
```

3/5 - empty

D code:

```
nothrow extern(C) Status LineRange_empty(LineRange * lr,
                                         int * empty) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");
    enforce(empty, "NULL 'empty' pointer.");

    *empty = lr.empty;
  });
}
```

3/5 - empty

D code:

```
nothrow extern(C) Status LineRange_empty(LineRange * lr,
                                         int * empty) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");
    enforce(empty, "NULL 'empty' pointer.");

    *empty = lr.empty;
  });
}
```

C header:

```
// mylibrary/mylibrary.h
#include <stdint.h> // For int32_t

Status LineRange_empty(LineRange range, int32_t * empty);
```

3/5 - empty

D code:

```
nothrow extern(C) Status LineRange_empty(LineRange * lr,
                                         int * empty) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");
    enforce(empty, "NULL 'empty' pointer.");

    *empty = lr.empty;
  });
}
```

C header:

```
// mylibrary/mylibrary.h
#include <stdint.h> // For int32_t

Status LineRange_empty(LineRange range, int32_t * empty);
```

C user example:

```
    int32_t empty = 0;
    status = LineRange_empty(lr, &empty);
```

4/5 - front

D code:

```
nothrow extern(C)
Status LineRange_front(LineRange * lr,
                       const(char) ** line) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");
    enforce(line, "NULL 'line' pointer.");

    *line = lr.front;
  });
}
```

4/5 - front

D code:

```
nothrow extern(C)
Status LineRange_front(LineRange * lr,
                       const(char) ** line) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");
    enforce(line, "NULL 'line' pointer.");

    *line = lr.front;
  });
}
```

C header:

```
// mylibrary/mylibrary.h

Status LineRange_front(LineRange range, const char ** value);
```

4/5 - front

D code:

```
nothrow extern(C)
Status LineRange_front(LineRange * lr,
                       const(char) ** line) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");
    enforce(line, "NULL 'line' pointer.");

    *line = lr.front;
  });
}
```

C header:

```
// mylibrary/mylibrary.h

Status LineRange_front(LineRange range, const char ** value);
```

C user example:

```
    const char * line = NULL;
    status = LineRange_front(lr, &line);
```

5/5 - popFront

D code:

```
nothrow extern(C) Status LineRange_popFront(LineRange * lr) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");

    lr.popFront();
  });
}
```

5/5 - popFront

D code:

```
nothrow extern(C) Status LineRange_popFront(LineRange * lr) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");

    lr.popFront();
  });
}
```

C header:

```
// mylibrary/mylibrary.h

Status LineRange_popFront(LineRange range);
```

5/5 - popFront

D code:

```
nothrow extern(C) Status LineRange_popFront(LineRange * lr) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");

    lr.popFront();
  });
}
```

C header:

```
// mylibrary/mylibrary.h

Status LineRange_popFront(LineRange range);
```

C user example:

```
    status = LineRange_popFront(lr);
```

Initializing the D runtime

If the program is not linked with a D compiler, the D runtime (the GC) must be initialized by the loading program.

Initializing the D runtime

If the program is not linked with a D compiler, the D runtime (the GC) must be initialized by the loading program.

Calling functions automatically *before* entering **`main()`**:

```
pragma (crt_constructor)
extern(C) int initialize() {        // Can have any name
  return rt_init();
}
```

Initializing the D runtime

If the program is not linked with a D compiler, the D runtime (the GC) must be initialized by the loading program.

Calling functions automatically *before* entering **`main()`**:

```
pragma (crt_constructor)
extern(C) int initialize() {        // Can have any name
  return rt_init();
}
```

Calling functions automatically *after* leaving **`main()`**:

```
pragma (crt_destructor)
extern(C) int terminate() {        // Can have any name
  return rt_term();
}
```

mylibrary.d

```d
import std;    // Importing the entire package for terseness.
import core.runtime;
import core.memory;

struct Status {
  int code;
  const(char) * errMsg;
}

// Function template that wraps extern(C) functions to translate potential Exceptions to Status objects.
nothrow Status tried(Func)(Func func, string functionName = __FUNCTION__) {
  try {
    func();
    return Status(0, "Success");

  } catch (Exception exc) {
    return Status(1, exc.msg.toStringz);

  } catch (Error err) {
    import core.stdc.stdio;
    import core.stdc.stdlib;
    fprintf(stderr, "\n%.*s(%zu): Failed to execute %.*s: %.*s\n",
            cast(int)err.file.length, err.file.ptr,
            err.line,
            cast(int)functionName.length, functionName.ptr,
            cast(int)err.msg.length, err.msg.ptr);

    abort();
  }

  assert(false);
}

// Function returning the "functionality" that our library will expose.
auto lineRange(string fileName = null) {
  enforce(!fileName.empty, "Empty file name.");

  return File(fileName)
         .byLine
         .map!strip
         .filter!(line => !line.empty)
         .filter!(line => !line.startsWith('#'));
}

// The struct that wraps the "functionality" of our library.
struct LineRange {
  alias LR = typeof(lineRange());
  LR lr;

  this(LR lr) {
    this.lr = lr;
    prime();
  }

  void prime() {
    if (lr.empty) {
      this.front = null;

    } else {
      this.front = lr.front.toStringz;
    }
  }

  auto empty() {
    return lr.empty;
  }

  const(char) * front;

  void popFront() {
    lr.popFront();
    prime();
  }
}
```

```d
// D runtime initialization
pragma (crt_constructor)
extern(C) int initialize() {
  return rt_init();
}

// D runtime deinitialization
pragma (crt_destructor)
extern(C) int terminate() {
  return rt_term();
}

// The library interface functions follow.

nothrow extern(C) Status LineRange_ctor(LineRange ** lr, const(char*) fileName) {
  return tried({
    enforce(lr, "NULL LineRange pointer.");
    enforce(fileName, "NULL file name.");

    *lr = new LineRange(lineRange(fileName.fromStringz.idup));
    GC.addRoot(*lr);
  });
}

nothrow extern(C) Status LineRange_dtor(LineRange * lr) {
  return tried({
    if (lr) {
      destroy(*lr);
      GC.removeRoot(lr);
    }
  });
}

nothrow extern(C) Status LineRange_empty(LineRange * lr, int * empty) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");
    enforce(empty, "NULL 'empty' pointer.");

    *empty = lr.empty;
  });
}

nothrow extern(C) Status LineRange_front(LineRange * lr, const(char) ** line) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");
    enforce(line, "NULL 'line' pointer.");

    *line = lr.front;
  });
}

nothrow extern(C) Status LineRange_popFront(LineRange * lr) {
  return tried({
    enforce(lr, "Uninitialized LineRange handle.");

    lr.popFront();
  });
}
```

mylibrary/mylibrary.h

```c
#pragma once

#include <stdint.h> // For int32_t

typedef struct {
  int32_t code;
  const char * errMsg;
} Status;

// The opaque handle type for the "functionality" of the library.
typedef void* LineRange;

// The constructor and the destructor.
Status LineRange_ctor(LineRange * range, const char * fileName);
Status LineRange_dtor(LineRange range);

// The InputRange interface exposed to C.
Status LineRange_empty(LineRange range, int32_t * empty);
Status LineRange_front(LineRange range, const char ** line);
Status LineRange_popFront(LineRange range);
```

deneme.c

An example user of the library:

```c
#include <stdio.h>
#include <mylibrary/mylibrary.h>

// Goes to 'finally' if the status code is non-zero.
#define bail_err()                                    \
  do {                                                \
    if (status.code) {                                \
      fprintf(stderr, "ERROR: %s\n", status.errMsg);  \
      ret = status.code;                              \
      goto finally;                                   \
    }                                                 \
  } while (0);

// Calls the specified function and bails if the call
// fails.
#define call(func, ...)                         \
  status = (func)(__VA_ARGS__);                 \
  bail_err();

int main(int argc, const char ** args) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s <file-name>\n", args[0]);
    return 1;
  }

  int ret = 0;
  Status status = {};
  LineRange lr = NULL;
```

```c
  call(LineRange_ctor, &lr, args[1]);

  while (1) {
    int empty = 0;
    call(LineRange_empty, lr, &empty);
    if (empty) {
      break;
    }

    const char * front = NULL;
    call(LineRange_front, lr, &front);
    printf("Printing on the C side: %s\n", front);

    call(LineRange_popFront, lr);
  }

finally:
  call(LineRange_dtor, lr);

  return ret;
}
```

Building

The D library:

```
$ dmd -shared mylibrary.d -oflibmylibrary.so
```

Building

The D library:

```
$ dmd -shared mylibrary.d -oflibmylibrary.so
```

The C program:

```
$ gcc deneme.c -Wl,-rpath=. libmylibrary.so -I. -odeneme
```

Executing

Reminder; the range object was:

```
File(fileName)
.byLine
.map!strip
.filter!(line => !line.empty)
.filter!(line => !line.startsWith('#'));
```

Executing

Reminder; the range object was:

```
          File(fileName)
          .byLine
          .map!strip
          .filter!(line => !line.empty)
          .filter!(line => !line.startsWith('#'));
```

The test file:

```
# myfile.txt

monday
   tuesday

 wednesday
```

Executing

Reminder; the range object was:

```
          File(fileName)
          .byLine
          .map!strip
          .filter!(line => !line.empty)
          .filter!(line => !line.startsWith('#'));
```

The test file:

```
# myfile.txt

monday
   tuesday

 wednesday
```

```
$ ./deneme myfile.txt
Printing on the C side: monday
Printing on the C side: tuesday
Printing on the C side: wednesday
```

Now we can call D from most other languages including Python.

Calling from Python

Opening the library with **`ctypes`**:

```python
from ctypes import *

mylibrary = cdll.LoadLibrary('libmylibrary.so')
```

Calling from Python

Opening the library with **`ctypes`**:

```
from ctypes import *

mylibrary = cdll.LoadLibrary('libmylibrary.so')
```

Python class corresponding to the library's **`Status`** struct:

```
class Status(Structure):
    _fields_ = [ ('code', c_int32),
                 ('errMsg', c_char_p) ]
```

Calling from Python

Opening the library with **`ctypes`**:

```python
from ctypes import *

mylibrary = cdll.LoadLibrary('libmylibrary.so')
```

Python class corresponding to the library's **`Status`** struct:

```python
class Status(Structure):
    _fields_ = [ ('code', c_int32),
                 ('errMsg', c_char_p) ]
```

Defining a callable (easier on the next slide):

```python
LineRange_ctor          = mylibrary.LineRange_ctor    # Magically locates the symbol
LineRange_ctor.restype  = Status                      # Sets the return type
LineRange_ctor.errcheck = check_status                # Sets an error checking function
```

Calling from Python

Opening the library with **`ctypes`**:

```
from ctypes import *

mylibrary = cdll.LoadLibrary('libmylibrary.so')
```

Python class corresponding to the library's **`Status`** struct:

```
class Status(Structure):
    _fields_ = [ ('code', c_int32),
                 ('errMsg', c_char_p) ]
```

Defining a callable (easier on the next slide):

```
LineRange_ctor          = mylibrary.LineRange_ctor    # Magically locates the symbol
LineRange_ctor.restype  = Status                      # Sets the return type
LineRange_ctor.errcheck = check_status                # Sets an error checking function
```

An error checking function:

```
def check_status(status, func, args):
    if status.code != 0:
        raise RuntimeError('Failed: {}'.format(status.errMsg.decode('utf-8')))
```

Calling from Python (continued)

Defining the callables can be easier with a function:

```python
def declare_func(func_str):
    func = eval('mylibrary.{}'.format(func_str))
    func.restype = Status
    func.errcheck = check_status
    return func

LineRange_ctor = declare_func('LineRange_ctor')
LineRange_dtor = declare_func('LineRange_dtor')
LineRange_empty = declare_func('LineRange_empty')
LineRange_front = declare_func('LineRange_front')
LineRange_popFront = declare_func('LineRange_popFront')
```

Calling from Python (continued)

Defining the callables can be easier with a function:

```python
def declare_func(func_str):
    func = eval('mylibrary.{}'.format(func_str))
    func.restype = Status
    func.errcheck = check_status
    return func

LineRange_ctor = declare_func('LineRange_ctor')
LineRange_dtor = declare_func('LineRange_dtor')
LineRange_empty = declare_func('LineRange_empty')
LineRange_front = declare_func('LineRange_front')
LineRange_popFront = declare_func('LineRange_popFront')
```

Python user example:

```python
lr = c_void_p()
fileName = 'myfile.txt'.encode('utf-8')
LineRange_ctor(byref(lr), fileName)
```

deneme.py

```python
from ctypes import *

mylibrary = cdll.LoadLibrary('libmylibrary.so')

class Status(Structure):
    _fields_ = [ ('code', c_int32),
                 ('errMsg', c_char_p) ]

def check_status(status, func, args):
    if status.code != 0:
        raise RuntimeError('Failed: {}'.format(status.errMsg.decode('utf-8')))

def declare_func(func_str):
    func = eval('mylibrary.{}'.format(func_str))
    func.restype = Status
    func.errcheck = check_status
    return func

LineRange_ctor = declare_func('LineRange_ctor')
LineRange_dtor = declare_func('LineRange_dtor')
LineRange_empty = declare_func('LineRange_empty')
LineRange_front = declare_func('LineRange_front')
LineRange_popFront = declare_func('LineRange_popFront')

lr = c_void_p()
fileName = 'myfile.txt'.encode('utf-8')
LineRange_ctor(byref(lr), fileName)

while True:
    empty = c_int32()
    LineRange_empty(lr, byref(empty))
    if empty.value != 0:
        break

    line = c_char_p()
    LineRange_front(lr, byref(line))
    print('Printing on the Python side: {}'.format(line.value.decode('utf-8')))

    LineRange_popFront(lr)

LineRange_dtor(lr)
```

Executing Python

```
$ LD_LIBRARY_PATH=. python3 deneme.py
Printing on the Python side: monday
Printing on the Python side: tuesday
Printing on the Python side: wednesday
```

Loading a D library from a D program

- **`dlopen`** cannot work; it does not know the D runtime.
- Must call **`loadLibrary`**.

```
import core.runtime;

  auto l = Runtime.loadLibrary("mylibrary.so");
```

"If the library contains a D runtime it will be integrated with the current runtime."

Conclusion

- `autowrap`[1] if using only from Python
- Otherwise
    - Work methodically to expose a C library interface
    - Use it from Python with `ctypes`

[1] https://github.com/symmetryinvestments/autowrap