Constructing a Repl(like) from scratch

Thoth Gunter


If we're all going to die,
let's at least enjoy it.

What to expect:

- Unsafe code
- Raw pointers
- Other unsavory things

Hope is that:

- Learn something interesting
- Some beauty will be laid bare

What is a REPL?

What is a REPL?

Read-Eval-Print-Loop

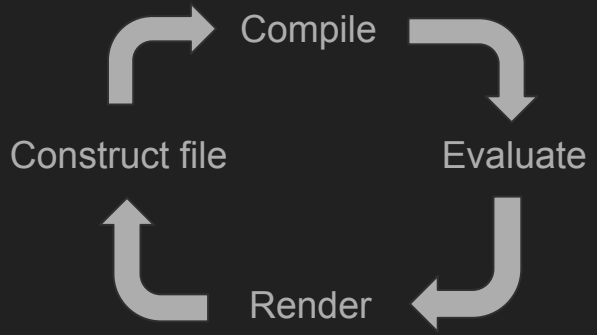
Read-Eval-Print-Loop

```
Python 3.6.0 (default, Jan 13 2017, 00:00:00)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
(test)jhaddad@jons-mac-pro ~$ ipython
Python 2.7.6 (default, Apr  9 2014, 11:48:52)
Type "copyright", "credits" or "license" for more information.

IPython 2.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details

In [1]: a = {}

In [2]: a.
a.clear       a.fromkeys    a.has_key     a.iteritems   a.itervalues
a.copy        a.get         a.items       a.iterkeys    a.keys

In [2]: a.
```

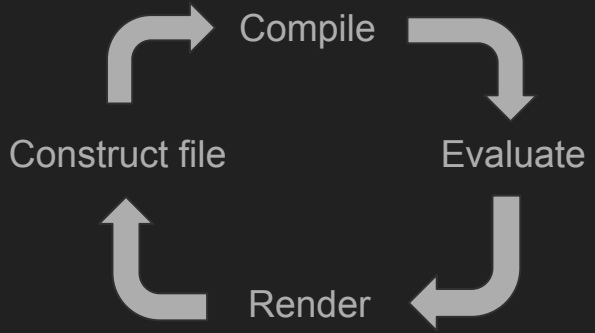
REPLs in compiled languages

- CINT (C interpreter)



REPLs in compiled languages

- CINT (C interpreter)



!!!User Beware!!!

State of the Rust REPL

- https://github.com/google/evcxr
- http://siciarz.net/24-days-of-rust-rusti/
- https://docs.rs/papyrus/0.11.1/papyrus/

But we’re not
making a REPL!

The general repl case VS my use case

GENERAL

- Refresh themselves on the language or library
- Quickly sketch a simple algorithm
- Iteratively build a program piece by piece testing the logic along the way

MY USE CASE

- Iteratively build a complex program
- Interactively operate on data
- Compare the results of complex algorithms when applied to the same state

The general repl case VS my use case

GENERAL

- Refresh themselves on the language or library
- Quickly sketch a simple algorithm
- Iteratively build a program piece by piece testing the logic along the way

MY USE CASE

- Iteratively build a complex program
- Interactively operate on data
- Compare the results of complex algorithms when applied to the same state

Goals/Sketch of the project

- Live update of function
- Easy to construct complex code
- Persistent memory
- Easy interface with any given code base.
- Cross (major) platform
  - BUT no browser (I want control of all my memory and all my bugs)

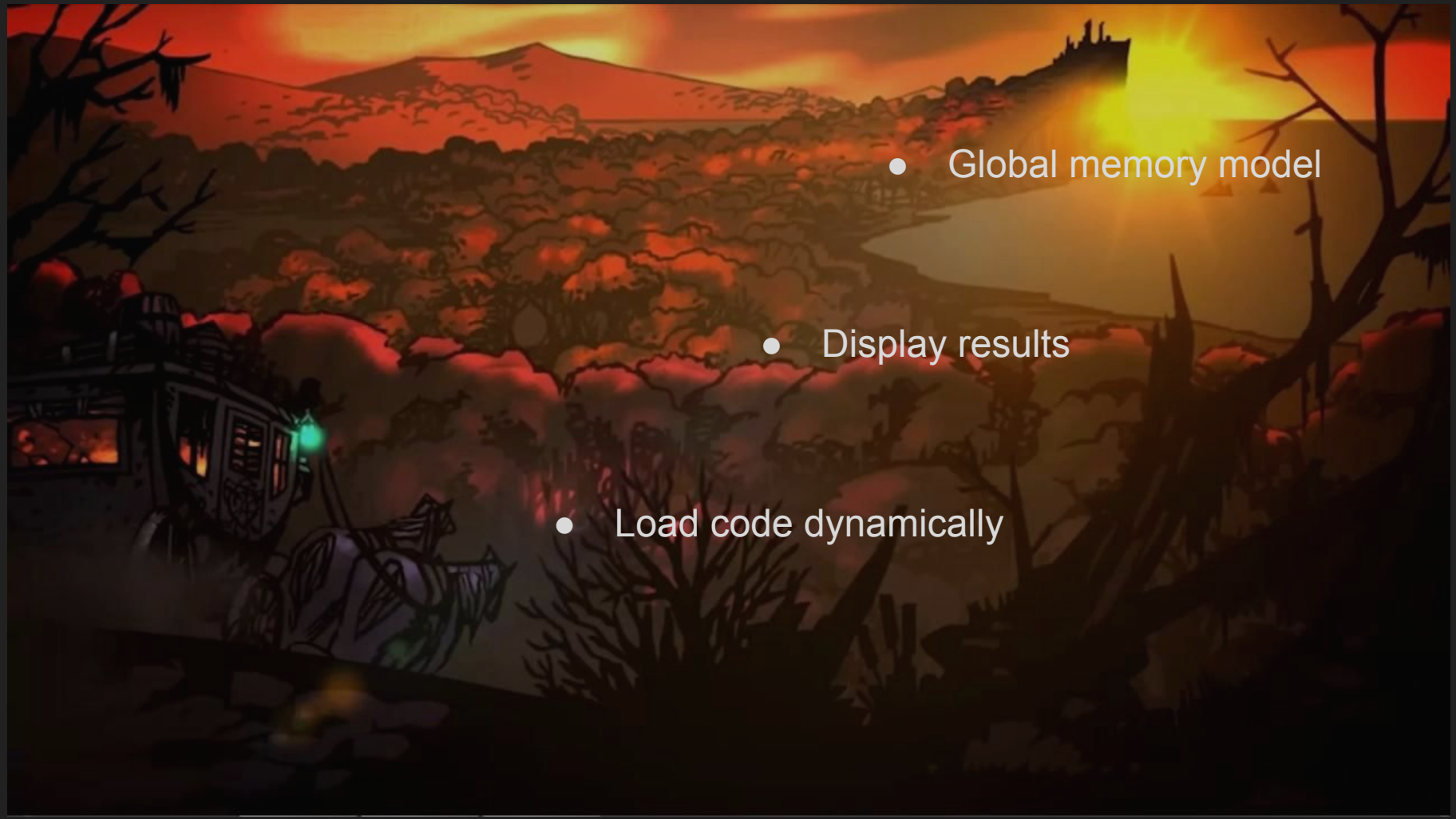
Goals/Sketch of the project

- Live update of function
- ~~Easy to construct complex code~~
  - Use editor of personal choice
- Persistent memory
- Easy interface with any given code base.
- Cross (major) platform
  - BUT no browser (I want control of all my memory)

- Global memory model
- Display results
- Load code dynamically

- Global memory model
- Display results
- Load code dynamically

- Global memory model
- Display results
- Load code dynamically



- Global memory model
- Display results
- Load code dynamically

Loading code… Hot code reloading

Loading code

dll(dynamic link library) [Windows]
so(shared object) [Unix]

Are formats that hold procedures and code that can be loaded during a programs run time or by multiple programs at once.

```
#![crate_type = "dylib"]  <= in src file
--crate-type=dylib     <= command line arg
crate-type = ["dylib"] <= toml arg
```

How does a program interact a dll or a so?

For ease of use:
https://github.com/szymonwieloch/rust-dlopen

Because I'm crazy and the above didn't exist when I started working on this.

- There are a number of os and kernel procedures programs. System header files are the way might know how to interact with these procedures if in C/C++.

Because we aren't using C/C++ we must define the function signature ourselves.

Find manual path for each of these

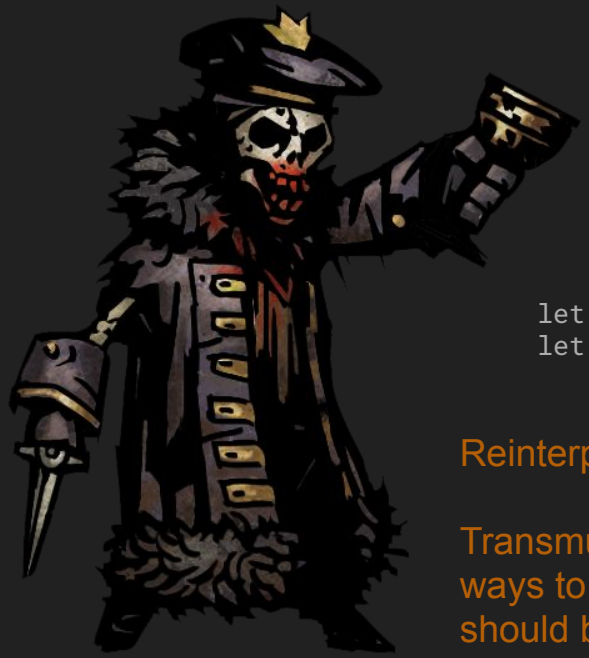
dll and so loading

```
#[cfg(target_os = "linux")]
    extern "C" {
        pub fn dlopen(filename: *const c_char, flag: c_int) -> *mut c_void;
        pub fn dlsym(lib_handle: *mut c_void, name: *const c_char) -> *mut c_void;
        pub fn dlclose(lib_handle: *mut c_void) -> c_int;
        pub fn dlinfo(lib_handle: *mut c_void, request: c_int, info: *mut c_void) -> c_int;
        pub fn dlerror() -> *mut c_char;
    }
```

```
#[cfg(target_os = "windows")]
    extern "C" {
        fn LoadLibraryA( path: *const i8 ) -> *mut c_void;
        fn GetProcAddress( lib: *mut c_void, name: *const i8 ) -> *mut c_void;
        fn FreeLibrary( lib: *mut c_void ) -> c_int;
        fn GetLastError() -> u32;
    }
```

```
let _fn = dlsym(shared_lib_handle, name);
let mut logic : fn()->f32 = unsafe{ std::mem::transmute( _fn ) };
```

```
let _fn = dlsym(shared_lib_handle, name);
let mut logic : fn()->f32 = unsafe{ std::mem::transmute( _fn ) };
```



```
let _fn = dlsym(shared_lib_handle, name);
let mut logic : fn()->f32 = unsafe{ std::mem::transmute( _fn ) };
```

Reinterprets the bits of a value of one type as another type.

Transmute is incredibly unsafe. There are a vast number of ways to cause undefined behavior with this function. Transmute should be the absolute last resort.

Interacting with our function

```
&mut RenderInstructions, &mut Storage, &InteractiveInfo
```

Inputs and return values

Interactivity code base

How do we know when to update code

Memory memory memory

Can you just box memory

No!!!!

How dll mem works

Custom alloc/dealloc/realloc

```
    pub struct GlobalStorage{
        pub storage: Vec<u8>,
        pub storage_filled: Vec<bool>,//TODO space footprint improvement use bits in u8
        reference: [TinyString;100],  //This is fixed size because I really want to stop myself from
over populating the global space
        stored_ptr: Vec<Ptr<GlobalStorage>>,
    }

    pub struct Ptr<S: Storage>{
        //TODO make backend_storage a Generic function that requires trait
        ptr: usize,
        type_hash: TypeId,
        backend_storage: *mut S
    }
    pub struct DyArray<T>{
        ptr: Ptr<GlobalStorage>,
        length: usize,
        capacity: usize,

        phantom: std::marker::PhantomData<T>,
    }
```

```
pub fn alloc<T: 'static>(&mut self, v: T)->Ptr<GlobalStorage>{unsafe{
    let size = std::mem::size_of::<T>();
    let src = (&v as *const T) as *const u8;

    let cursor = self.storage.len();
    for i in 0..size{
       //TODO
       //SLOW SPEED ME UP
       //I don't think I want to be pushing every thiem like this
       //TODO
       //byte alignments
       self.storage.push(*src.offset(i as isize));
       self.storage_filled.push(true);
    }
    return Ptr{ ptr: cursor, type_hash: TypeId::of::<T>(), backend_storage: self as *mut _};
}}
```

How about a demo?