Interactive C++ Compilation (REPL): The Lean Way

by **Viktor Kirilov**

About me

- my name is Viktor Kirilov - from Bulgaria
- 4 years of professional C++ in the games / VFX industries
- working on personal projects since 01.01.2016 (2+ years)
- some consulting and contract work

Tools of the trade

- compilers: Visual Studio, GCC, Clang, Emscripten
- tools: CMake, Python, git, clang-format, valgrind, sanitizers
- services: GitHub, Travis CI, AppVeyor

Passionate about

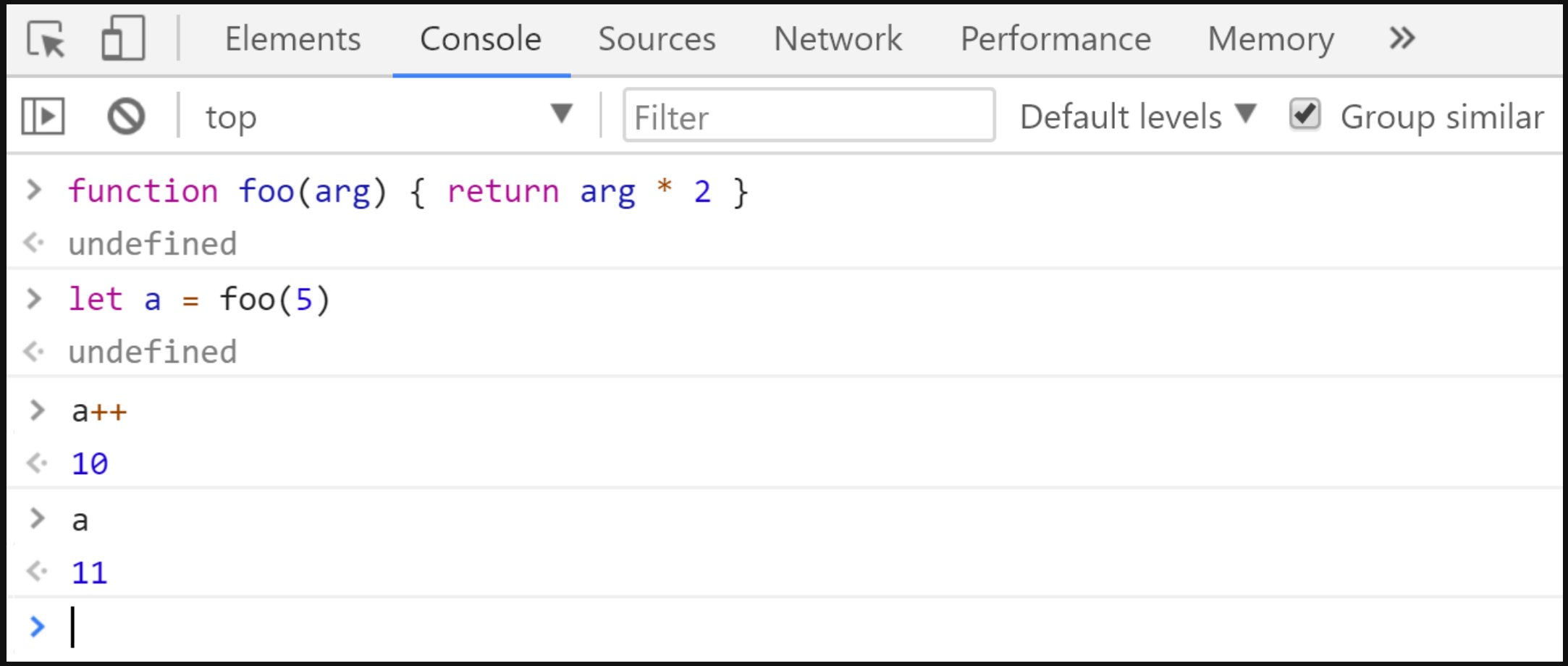
- game development and game engines
- data-oriented design and HPC
- build systems and good software development practices
- cryptocurrencies and blockchain

This presentation

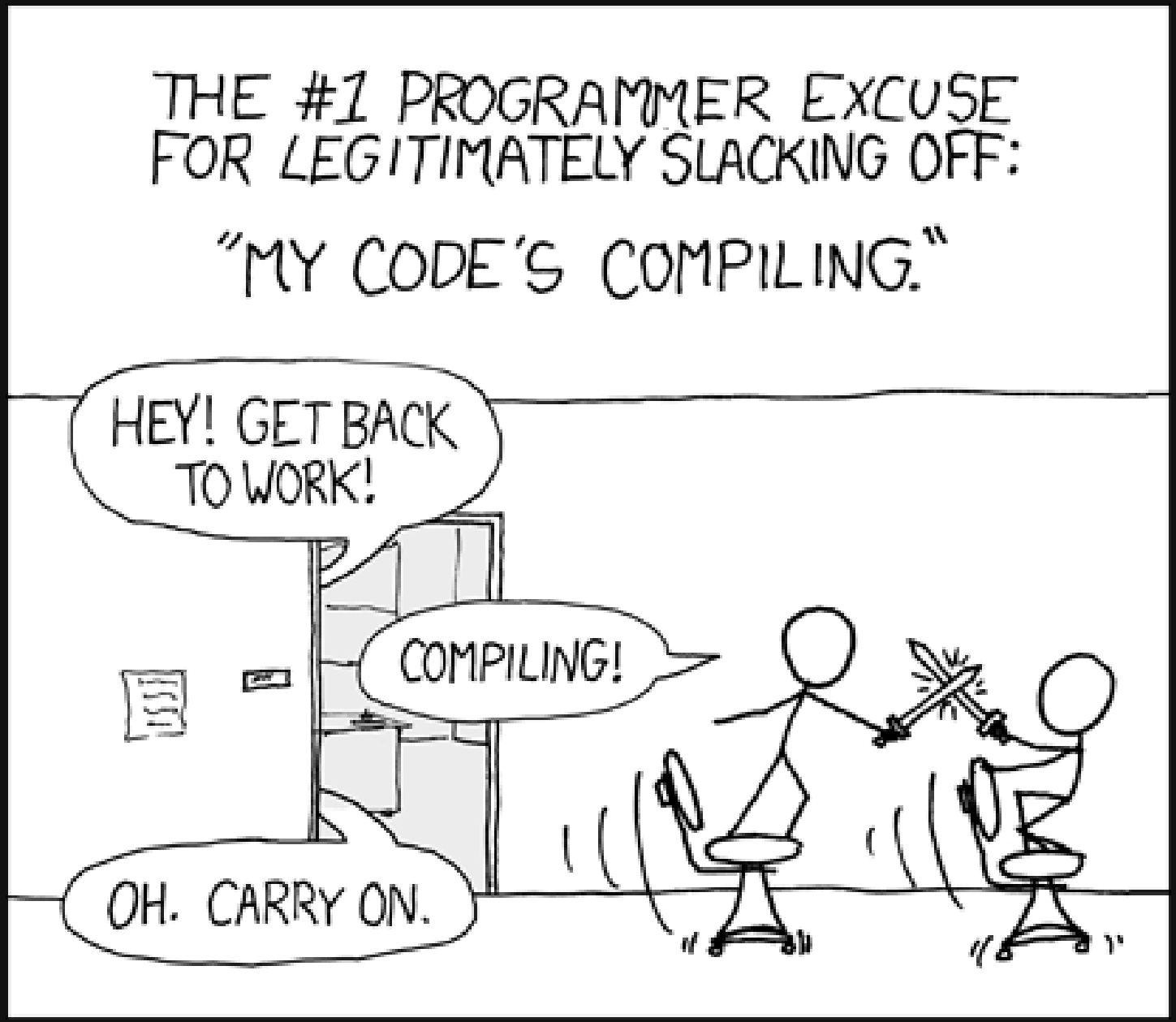
- Introduction
- Demo (danger: live)
- How stuff works
- About the demo project
- How to integrate
- Room for improvement
- Q&A

What is a REPL

- interpreted languages (JavaScript, Python, etc.)
- consoles/shells - cmd.exe, bash
- can iteratively append statements and definitions



Major C++ issue - compile times



The need for runtime compilation

Improved workflow

- no need to restart the program - can preserve state
  - faster iteration times
  - less need for a scripting language
    - no binding layer
    - no need for a virtual machine
    - code in one language
  - debuggers aren't infinitely powerful for inspection
  - can hack something quickly
- compile times are probably optimized for what is hot-reloadable

Runtime Compilation in C++

- loading shared libraries OR hot-patching - entire functions are replaced
  - usually quite intrusive (interfaces, constraints, complicated setup)
  - in game engines: Unreal, others...
  - hot-patching (with very little setup): Live++, Recode
  - Visual Studio "Edit & Continue" - 0 setup, but limited
- REPL-like (mix global and function scopes and append to them)
  - cling - by researchers at CERN - built on top of LLVM
    - inspector, Jupiter
    - hard to integrate in a platform/compiler agnostic way
  - the technique from this presentation (RCRL)
- http://bit.ly/runtime-compilation-alternatives

The demo host application API

```
#ifdef HOST_APP
#define HOST_API SYMBOL_EXPORT // __declspec(dllexport)
#else // plugin imports
#define HOST_API SYMBOL_IMPORT // __declspec(dllimport)
#endif

class HOST_API Object {
    float m_x = 0, m_y = 0, m_r = 0.3f, m_g = 0.3f, m_b = 0.3f;
    float m_rot = 0, m_rot_speed = 1.f;

    friend HOST_API Object& addObject(float x, float y);
    Object() = default;

public:
    void translate(float x, float y);
    void colorize(float r, float g, float b);
    void set_speed(float speed);

    void draw();
};

HOST_API std::vector<Object>& getObjects();
HOST_API Object& addObject(float x, float y);
```

Demo - RCRL: Read-Compile-Run-Loop

3 section types - global, vars, once

```
// global
int foo() { return 42; }

// vars
int a = foo();
auto& b = a;

// once
a++;

// global
#include <iostream>
void print() { std::cout << a << b << std::endl; }

// once
print(); // =======> will result in "4343" being printed
```

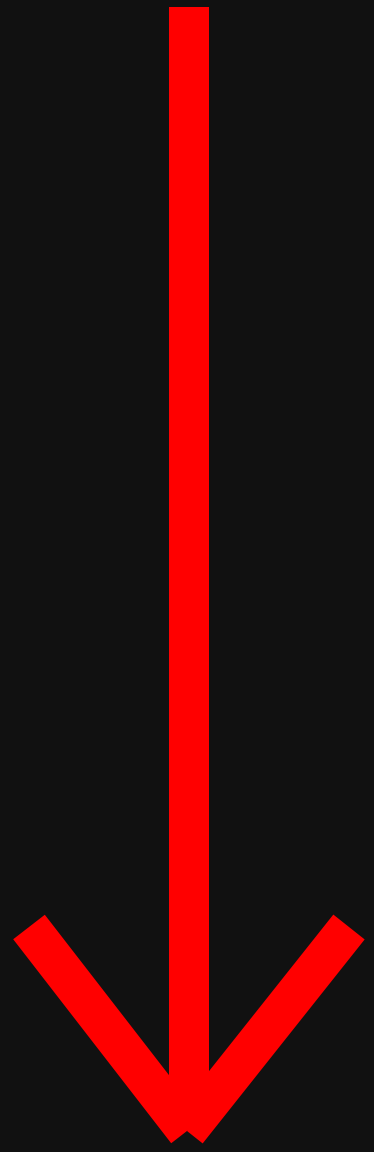
How it works

- submit code
- reconstruct a .cpp file from sections in the proper order
  - include all global and vars sections (+ from the past)
  - use once sections only from the current submission
- compile the .cpp file as a shared library (plugin)
  - link against the host application
  - do not proceed if compilation fails
- copy the resulting plugin with a new name
- load the copy (without unloading old ones)
  - initializes globals top-to-bottom
    - executes once statements as part of that step
    - initializes persistent variables from vars sections

How it works

- cleanup:
  - calls the destructors of variables in vars sections in reverse order
  - unloads the plugins in reverse order
    - calls the destructors of variables in global sections
    - deletes the plugins from the filesystem
- variables can be easily put in global and once sections too
  - in global - re-initialized on every recompilation + side effects
  - in once - available only within the section
- compilation is done in a background process so it isn't blocking
- object contents aren't printed by default (unlike traditional REPLs)

What the .cpp file looks like - global, once



```
// global
#include <iostream>
// once
std::cout << "hello!";
```

submitted code

```
#include "path/to/rcrl_for_plugin.h"

#include <iostream>

RCRL_ONCE_BEGIN
std::cout << "hello!";
RCRL_ONCE_END
```

.cpp file

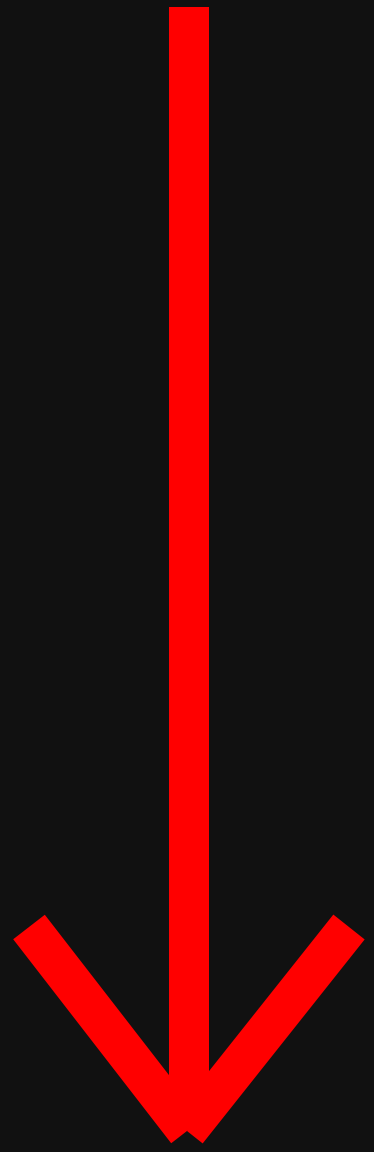
```
#include "path/to/rcrl_for_plugin.h"

#include <iostream>

static int rcrl_anon_12 = []() {
std::cout << "hello!";
return 0; }();
```

expanded macros

What the .cpp file looks like - vars, once



```
// vars
int a = 5;
// once
a++;
```

submitted code

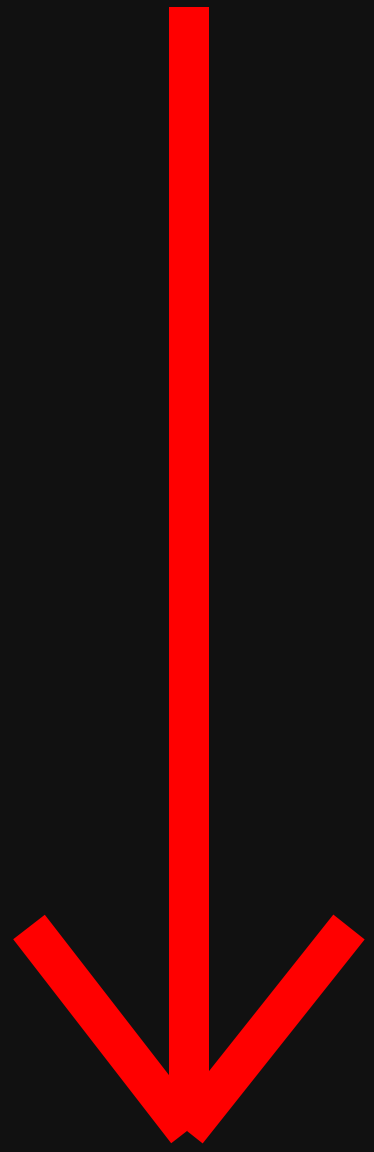
```
#include "path/to/rcrl_for_plugin.h"

static int& a = *[]() {
    auto& address = rcrl_get_persistence("a");
    if(address == nullptr) {
        address = (void*)new int(5);
        rcrl_add_deleter(address, [](void* ptr)
            { delete static_cast<int*>(ptr); });
    }
    return static_cast<int*>(address);
}();

static int rcrl_anon_12 = []() {
a++;
return 0; }();
```

expanded macros

What the .cpp file looks like - auto in vars



```
// vars
auto a = 5;
```

submitted code

```
#include "path/to/rcrl_for_plugin.h"

static auto type_of_a = []() -> auto { // use with decltype() to deduce
    auto temp = (5);
    return temp;
};
static decltype(type_of_a())& a = *[]() {
    auto& address = rcrl_get_persistence("a");
    if(address == nullptr) {
        address = (void*)new decltype(type_of_a())(5);
        rcrl_add_deleter(address, [](void* ptr)
            { delete static_cast<decltype(type_of_a())*>(ptr); });
    }
    return static_cast<decltype(type_of_a())*>(address);
}();
```

expanded macros

Linking to the host application

- for interacting with the host application through the exported API
- the .cpp file always includes a header called "rcrl_for_plugin.h"
  - with helper macros
  - with symbols exported from the host app for persistence

```
RCRL_SYMBOL_IMPORT void*& rcrl_get_persistence(const char* var_name);
RCRL_SYMBOL_IMPORT void   rcrl_add_deleter(void* address, void (*deleter)(void*));
```

- set the ENABLE_EXPORTS CMake property to true on executables

```
set_target_properties(my_executable PROPERTIES ENABLE_EXPORTS ON)
```

- no symbols are exported by default on Windows
  - unlike Unix (with no "-fvisibility=hidden")
  - WINDOWS_EXPORT_ALL_SYMBOLS target property (CMake)
- can also link to shared objects instead of executables

Explicit symbol exports

```
#if defined _WIN32 || defined __CYGWIN__
#define SYMBOL_EXPORT __declspec(dllexport)
#define SYMBOL_IMPORT __declspec(dllimport)
#else
#define SYMBOL_EXPORT __attribute__((visibility("default")))
#define SYMBOL_IMPORT
#endif

#ifdef DLL_EXPORTS // if this is defined - the API is exported
#define MY_API SYMBOL_EXPORT
#else
#define MY_API SYMBOL_IMPORT
#endif

MY_API void bar();

class MY_API MyClass {
    // everything here is exported
};
```

Explicit symbol exports

Pros:

- improves link time
- circumvents the GOT (Global Offset Table) for calls => profit
- load times also improved (extreme templates case: 45 times)
- reduces the size of your DSO by 5-20%
- https://gcc.gnu.org/wiki/Visibility

Cons:

too much work if only for the RCRL technique

Parser - vars section

- parses the type, name and initializer for variable definitions
- tiny - less than 400 lines of code
- supports complex constructs
  - templates
  - decltype
  - complex initializers
  - references (allocates a pointer to T instead of the type T)
  - auto (lambda + "decltype()" to get the deduced type)

```
std::map<decltype(i_return_int()), std::vector<std::string>> vec = {{5, {}}, {6, {}}
```

- only variable definitions are allowed

Restrictions - vars section

- [*] no C arrays and "alignas()" - use "std::array<>" instead
- [*] one definition per statement (cannot handle "type A, B;")
- [*] don’t use "auto*" - let auto deduce pointers
- [*] raw string literals will mess things up
- stateful lambdas only to "std::function<>" - cannot use auto
  - stateless can be with auto in global sections
    - but then you can use normal functions...
- no deleted operator new/delete for types
- no rvalue references as variables
- const references don't extend the lifetime of temporaries

the ones with [*] can be lifted by improving the parser

Other restrictions and notes

- don’t rely on the address of functions - changes after recompilation
- don't use the static keyword (either in function or global scope)
- no goto between once sections
- "decltype()" of names from vars sections => reference to the type
- constexpr variables should go in global sections and not in vars
- don't use the preprocessor in vars sections (and limit it in the others)
- global variables will be re-initialized on each recompilation without destroying the old versions - and side effects will accumulate
  - that is what the vars sections are for
- class static member vars go in global - re-initialized on recompilation

The benefits are worth the price of these restrictions

RCRL API

```
// rcrl.h

enum Mode {
    GLOBAL,
    VARS,
    ONCE
};

std::string cleanup_plugins(bool redirect_stdout = false);

bool submit_code(std::string code, Mode default_mode = ONCE, ...);

std::string get_new_compiler_output();

bool is_compiling();

bool try_get_exit_status_from_compile(int& exitcode);

std::string copy_and_load_new_plugin(bool redirect_stdout = false);
```

Sample loop

```
while(true) {
    if(submit() && is_compiling() == false) {
        editor.lock();
        submit_code(editor.code());
        compiler_output.clear();
    }

    compiler_output += get_new_compiler_output();

    int status;
    if(try_get_exit_status_from_compile(status)) {
        if(status == 0) {
            // on success - append to history and clear editor
            history += editor.code();
            editor.code().clear();
            copy_and_load_new_plugin();
        }
        editor.unlock();
    }
}
```

The repository

- RCRL core - requires C++11 (for auto variables - C++14)
  - 5 source files - in /src/rcrl/
    - rcrl.h (66 loc) - the main API - consists of 6 forward declarations
    - rcrl.cpp (274 loc) - the engine
    - rcrl_for_plugin.h (55 loc) - for use by the plugin
    - rcrl_parser.h (30 loc)
    - rcrl_parser.cpp (368 loc) - parses sections and variables
  - dependency on /src/third_party/tiny-process-library - for processes
- the rest of the demo - GUI, exported API, third party libraries
  - /src/third_party/glfw - for the window, OpenGL context and input
  - /src/third_party/imgui - for the GUI
  - /src/third_party/ImGuiColorTextEdit - imgui text editor extension

RCRL CMake integration

- CMake - easy - the demo works on all platforms with any compiler
- assumes the plugin is part of the whole CMake tree (not optimal !!!)
- a precompiled header is used - fill it with common includes
- expected preprocessor identifiers (easily setup in CMake)
  - RCRL_PLUGIN_FILE - full path to the .cpp file for the plugin
  - RCRL_PLUGIN_NAME - the plugin CMake target
  - RCRL_BUILD_FOLDER - root CMake build folder
  - RCRL_BIN_FOLDER - where the compiled plugin goes
  - RCRL_EXTENSION - .dll/.so/.dylib
  - RCRL_CONFIG - only for Visual Studio, XCode...

How to integrate properly

- the RCRL core (rcrl.cpp) is expected to be modified for custom needs
  - not trying to be a one-size-fits-all solution
  - no need for CMake - can call the build system directly
    - ninja is exceptionally fast compared to make/msbuild
    - can even call the compiler directly
  - requires knowledge of build systems, compilers, linking, etc.
- try to avoid linking against big static libraries
- disabling optimizations will shorten the build time
- the editor may be separate from the application
  - easier support for auto-completion and syntax highlighting
  - code can be sent through sockets or some other way
  - even the RCRL engine may be separate from the application
    - the application just needs to load the new plugins

Room for improvement of the engine

- global and vars sections can be merged - need a better parser
  - we might even get rid of once sections with the help of LibClang
- auto complete - probably with LibClang
- crash handling when loading the plugins after compilation
- compiler error messages
  - mapping between lines of the submitted code and in the .cpp
- debugging
  - ability to set breakpoints? just a thought...
- what was mentioned so far in the restrictions and integration parts

Random thoughts

- this technique can be used for other compiled languages too
- perhaps resumable functions might help with:
  - the need to allocate the types in vars sections
  - extending the lifetime of temporaries with a const reference
- applications can have an optional module which enables live hacking
  - a module implementing this technique (the RCRL engine)
  - a C++ compiler (the same version used for the application)
  - application API headers with dll exported symbols
  - application export lib so it can be linked to (Windows only)

Q&A

- Slides: https://slides.com/onqtam/2018_interactive_cpp_compiler
- Demo Project: https://github.com/onqtam/rcrl
- Alternatives: http://bit.ly/runtime-compilation-alternatives
- Blog: http://onqtam.com
- GitHub: https://github.com/onqtam
- Twitter: https://twitter.com/KirilovVik
- E-Mail: vik.kirilov@gmail.com