Инфраструктура LLVM

Лекция №5

Теория языков программирования

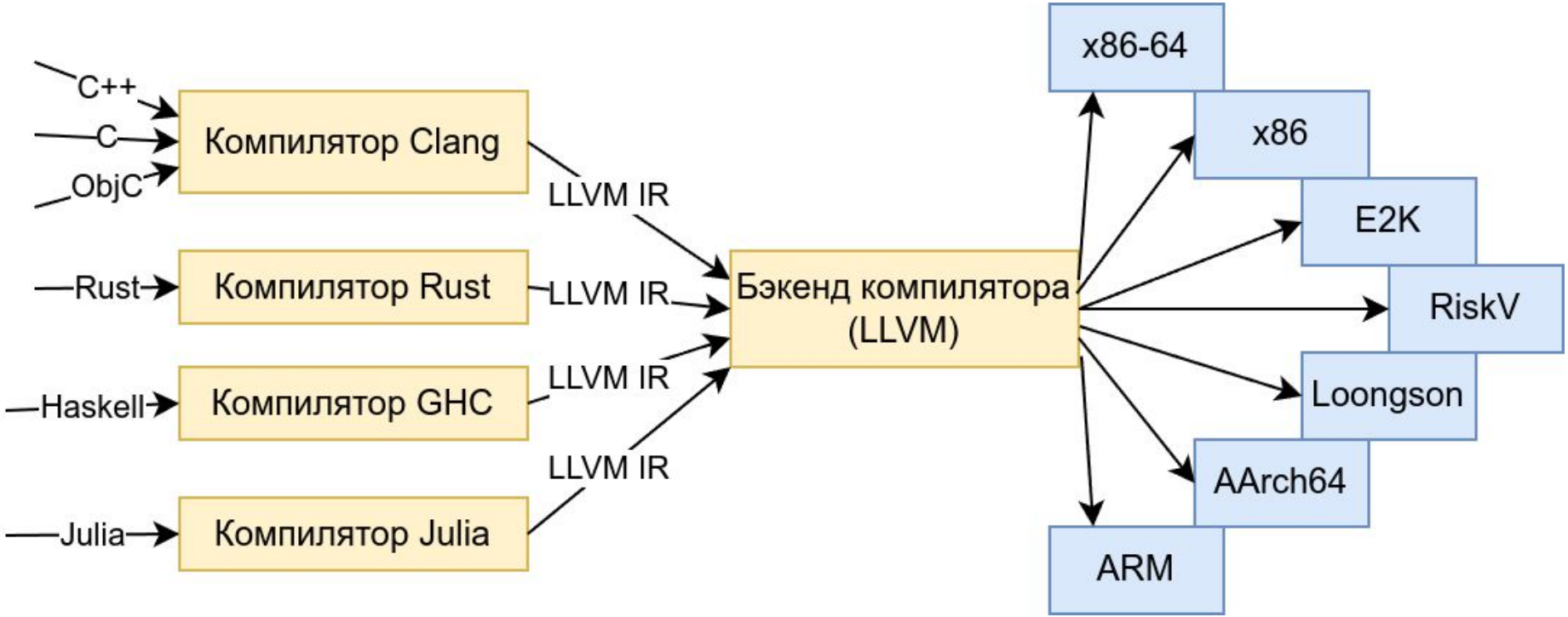
Вопросы этой лекции

Допустим, мы хотим использовать LLVM для разработки компилятора:

1. Что такое LLVM?
2. Как покрыть компилятор тестами?
3. Как написать компилятор?

Что такое LLVM

Применение LLVM



Другие применения LLVM

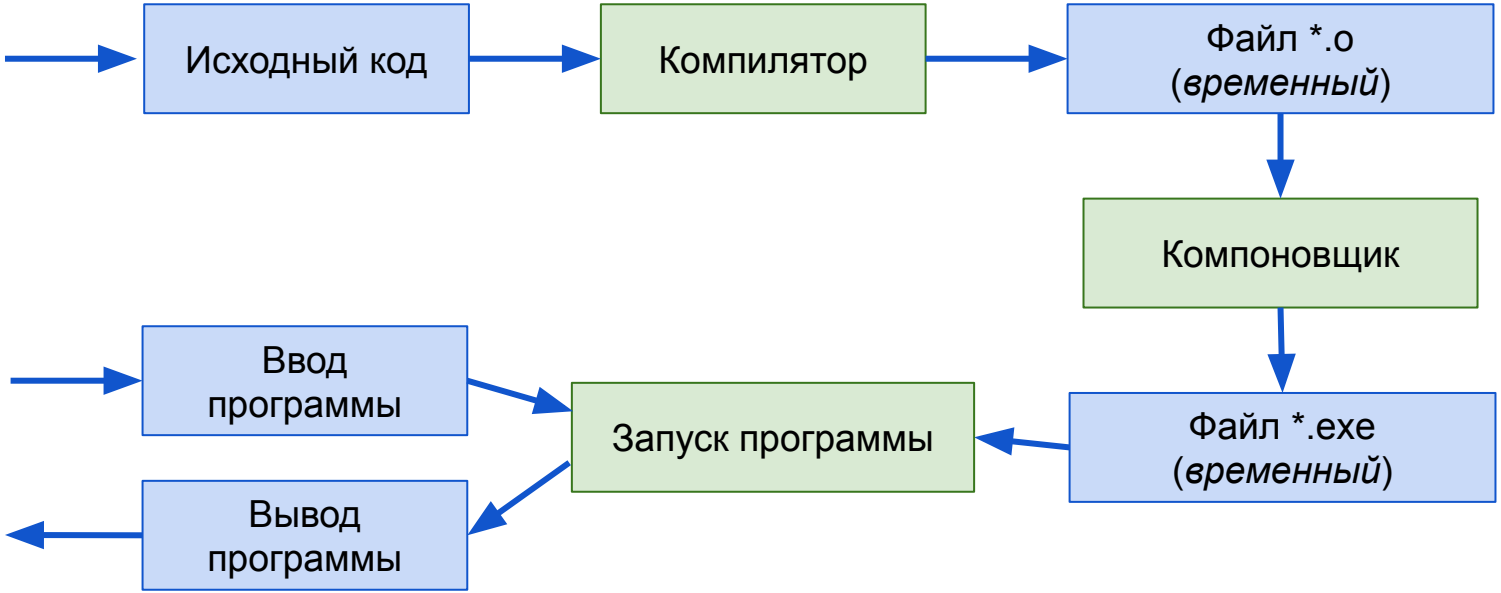
1. Компиляция шейдеров OpenGL
2. Компиляция CUDA / OpenCL — вычисления на GPU
3. Фреймворки машинного обучения

Преимущества LLVM

1. Хороший API генерации кода
2. LLVM IR, включая MLIR — Multi-Level IR
3. Большой набор готовых оптимизаций
4. Инструментирование
    - Отладочная информация
    - Sanitizers
    - Code Coverage
    - Fuzzing

Тесты компилятора

Первый тест компилятора



Catch3 для тестов

```
// clang-format off
TEST_CASE("Can run echo", "[process]")
{
    support::Process process("echo");
    process.addArgument("Moscow");
    process.runAndCheck();

    REQUIRE(process.stdoutContents() == "Moscow\n");
}
// clang-format on
```

Параметризация теста

```
// Это параметризованный тест, testCase хранит текущий сценарий.
auto testCase = GENERATE(
    InputOutputTestCase{
        "features/input_output/getchar_5_times.tig",
        "Hello, World!",
        "Hello"
    }
);
// Устанавливаем описание текущего сценария.
CAPTURE(testCase.sourcePath);
```

Тест запуска программы

```
auto executableFile = compileToTempFile(testCase.sourcePath);
std::string stdout = runProgramWithCheck(
    executableFile.path(),
    testCase.input
);
REQUIRE(stdout == testCase.exectedOutput);
```

Прикладные вопросы

Вопрос: как в языке C++23

1. Создать временный файл
2. Найти исполняемый файл в PATH
3. Запустить дочерний процесс

Прикладные вопросы

Вопрос: как в языке C++23

1. Создать временный файл
2. Найти исполняемый файл в PATH
3. Запустить дочерний процесс

Варианты:

1. Использовать std::filesystem и boost::process

Прикладные вопросы

Вопрос: как в языке C++23

1. Создать временный файл
2. Найти исполняемый файл в PATH
3. Запустить дочерний процесс

Варианты:

1. Использовать std::filesystem и boost::process
2. Использовать API из LLVM::Support

Подключение LLVM к CMake

Интеграция с CMake

```
find_package(LLVM 22 REQUIRED CONFIG)
find_package(zstd 1 REQUIRED)

# Подключаем CMake-модули LLVM
list(APPEND CMAKE_MODULE_PATH ${LLVM_CMAKE_DIR})
include(AddLLVM)
```

Подключение модулей LLVM к библиотеке

```
# Подключаем библиотеки LLVM
llvm_map_components_to_libnames(LLVM_LIBS Core Support)
target_link_libraries(
  libsupport PRIVATE ${LLVM_LIBS}
)
target_include_directories(
  libsupport PRIVATE ${LLVM_INCLUDE_DIRS}
)
```

Объекты LLVM

Модуль — llvm::Module

- Содержит функции, глобальные переменные, константы
- Нужен для раздельной компиляции

Функция — llvm::Function

- Имеет имя
- Имеет сигнатуру
- Может не иметь тела

Создание:

```
llvm::FunctionType::get(_int32Type, {_pointerType}, true);
```

Вызов:

```
_builder.CreateCall(_printfFn, {format, text});
```

Базовый блок инструкций — llvm::BasicBlock

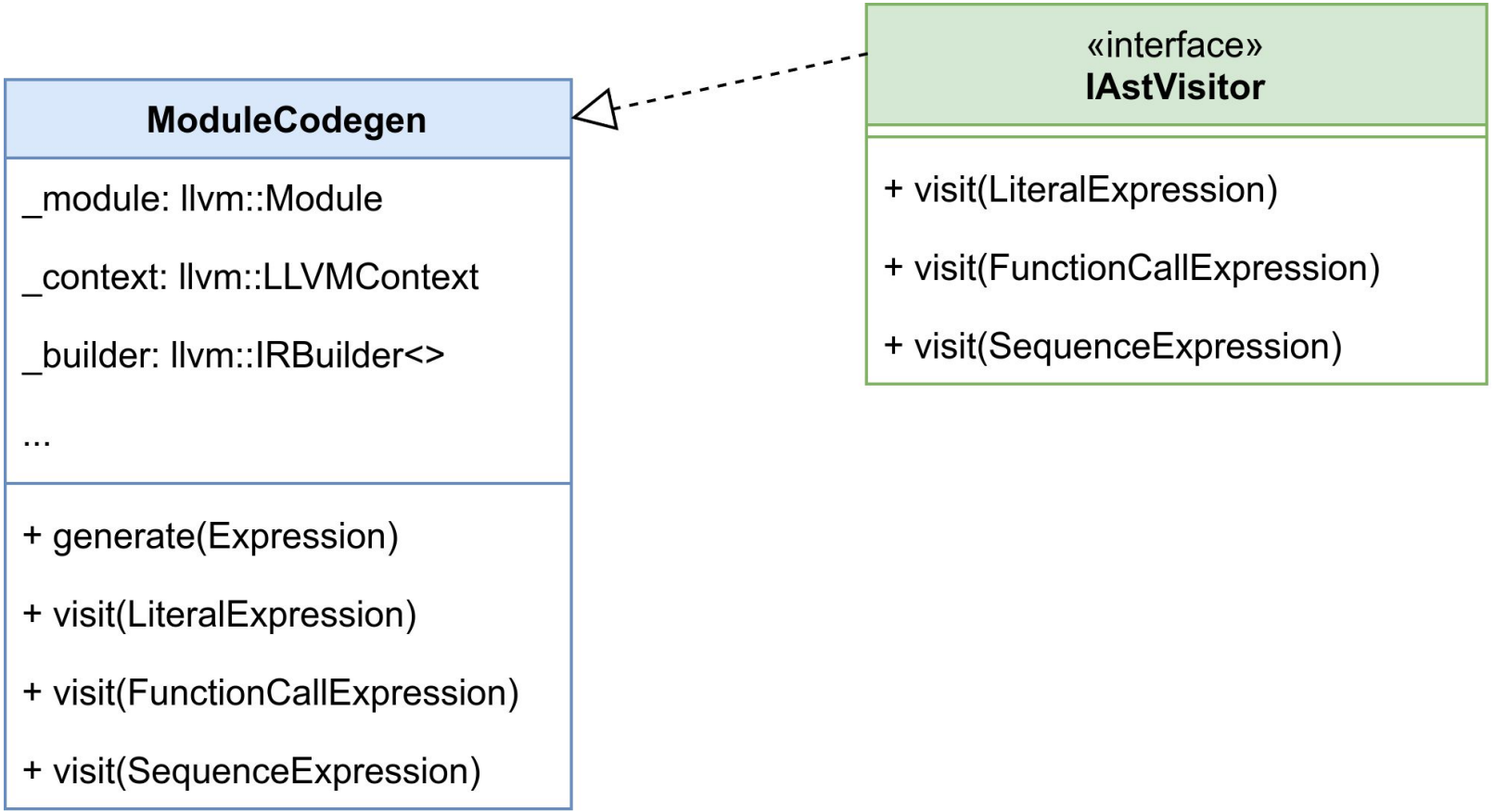
- Функция состоит из BasicBlock
- Каждый BasicBlock в LLVM:
  - содержит инструкции
  - завершается ровно одной терминальной инструкцией
  - может совершать безусловный переход в целевой BasicBlock
  - может совершать условный переход в два целевых BasicBlock

Создание:

```
llvm::BasicBlock::Create(_context, "entry", main)
```

Генерация LLVM IR

Класс ModuleCodegen



Класс llvm::IRBuilder<>

Вставляет инструкции в указанный базовый блок.

```
llvm::BasicBlock* entry = llvm::BasicBlock::Create(
    _context, "entry", main
);
_builder.SetInsertPoint(entry);

llvm::Constant* exitCode = llvm::ConstantInt::getSigned(
    _int32Type, 0
);
_builder.CreateRet(exitCode);
```

Метод ModuleCodegen::generate

```
llvm::Function* main = declareMainFunction();
// Создаём начальный блок и начинаем вставку инструкций в него.
llvm::BasicBlock* entry = llvm::BasicBlock::Create(
    _context, "entry", main
);
_builder.SetInsertPoint(entry);
// Выполняем обход дерева для генерации кода
program.accept(*this);
// Завершаем функцию main,
addExitToMainFunction();
// Проверяем LLVM IR в созданной функции.
verifyFunction(*main);
```

Валидация созданного LLVM IR

```
void ModuleCodegen::verifyFunction(llvm::Function& fn) {
    std::string output;
    llvm::raw_string_ostream outputStream(output);
    if (llvm::verifyFunction(fn, &outputStream)) {
        throw std::logic_error(std::format(
          "Function '{}' verification failed: {}",
          fn.getName().str(), output
        ));
    }
}
```

Интеграция с libc

Интеграция с libc

Задача: реализовать ввод, вывод и завершение программы

Интеграция с libc

Задача: реализовать ввод, вывод и завершение программы

Решение:

printf("%s", value) — для вывода строк

printf("%d", value) — для вывода чисел

fflush(stdout) — flush

getchar() — посимвольный ввод

exit(value) — завершение программы

Вызов функций libc

```
/**
* Генерирует вызов `printf("%s", text)`.
*/
void ModuleCodegen::generatePrintCall()
{
    llvm::Value* format = addStringLiteral("%s");
    llvm::Value* text = _values.pop();

    _builder.CreateCall(_printfFn, {format, text});
}
```

Инициализация функций libc

```
// Функция i32 printf(i8 *format, ...);
llvm::FunctionType* fnType = llvm::FunctionType::get(
    _int32Type, {_pointerType}, true
);
_printfFn = llvm::Function::Create(
   fnType,
   llvm::Function::ExternalLinkage,
   "printf",
   _module
);
```

Как выглядит LLVM IR

```
declare i32 @printf(ptr noundef, ...) #1

define dso_local i32 @main() #0 {
 %1 = call i32 (ptr, ...) @printf(
    ptr noundef @.str, ptr noundef @.str.1
 )
 %2 = call i32 (ptr, ...) @printf(
    ptr noundef @.str.2, i32 noundef 49
 )
 ret i32 0
}
```

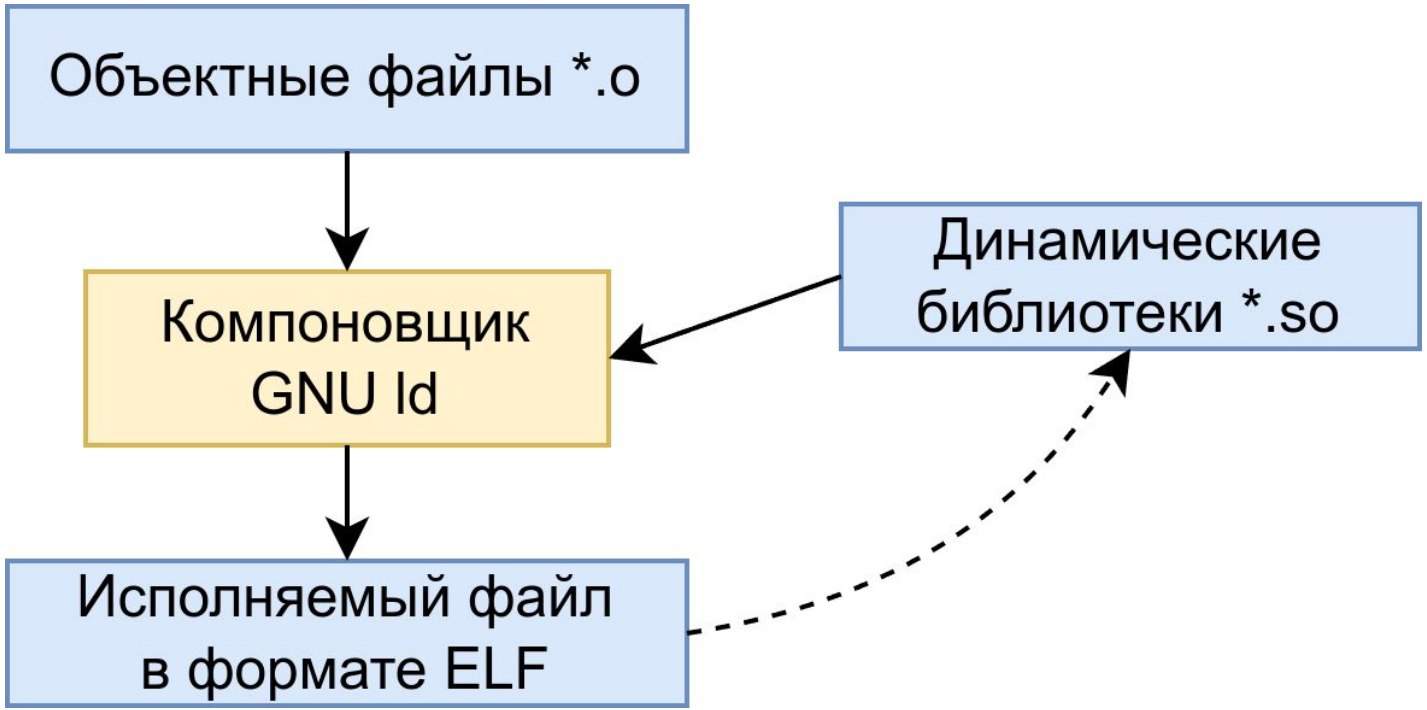
Генерация объектного файла

Генерация объектного файла

```
/**
* Преобразует промежуточный код в объектный файл с машинным кодом
целевой платформы:
*  - принимает на вход модуль LLVM IR;
*  - выполняет оптимизации и затем генерацию машинного кода
средствами LLVM.
*/
void generateObjectFile(
   llvm::Module& module,
   bool applyOptimizations,
   const std::filesystem::path& outputPath);
```

Компоновка

Что такое компоновка



Компоновка для Linux с помощью GNU ld

Прямой вызов ld:

```
ld -z relro --hash-style=gnu --eh-frame-hdr -m elf_x86_64
-dynamic-linker /lib64/ld-linux-x86-64.so.2 -o program.exe
/usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/11/crtbegin.o
-L/usr/lib/gcc/x86_64-linux-gnu/11 -L/lib/x86_64-linux-gnu
-L/usr/lib/x86_64-linux-gnu -L/lib -L/usr/lib program.o -lgcc
--as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s
--no-as-needed /usr/lib/gcc/x86_64-linux-gnu/11/crtend.o
/usr/lib/x86_64-linux-gnu/crtn.o
```

Компоновка для Linux с помощью GCC

Вызов драйвера GCC:

```
gcc program.o -o program.exe
```

Программный вызов компоновки

```
support::Process process("gcc");
process.addArgument(objectFilePath.string());
process.addArgument("-o");
process.addArgument(outputPath.string());
process.runAndCheck();
```

Как спросить ИИ-модель про LLVM

Пример промтпа

Как с помощью **LLVM 22** на **C++23** сгенерировать код **LLVM IR**, вызывающий `**fflush(stdout)**`?

Пример промтпа

Напиши на **C++23** с использованием **LLVM 22** пример кода, который генерирует код на **LLVM IR**, выполняющий следующее:

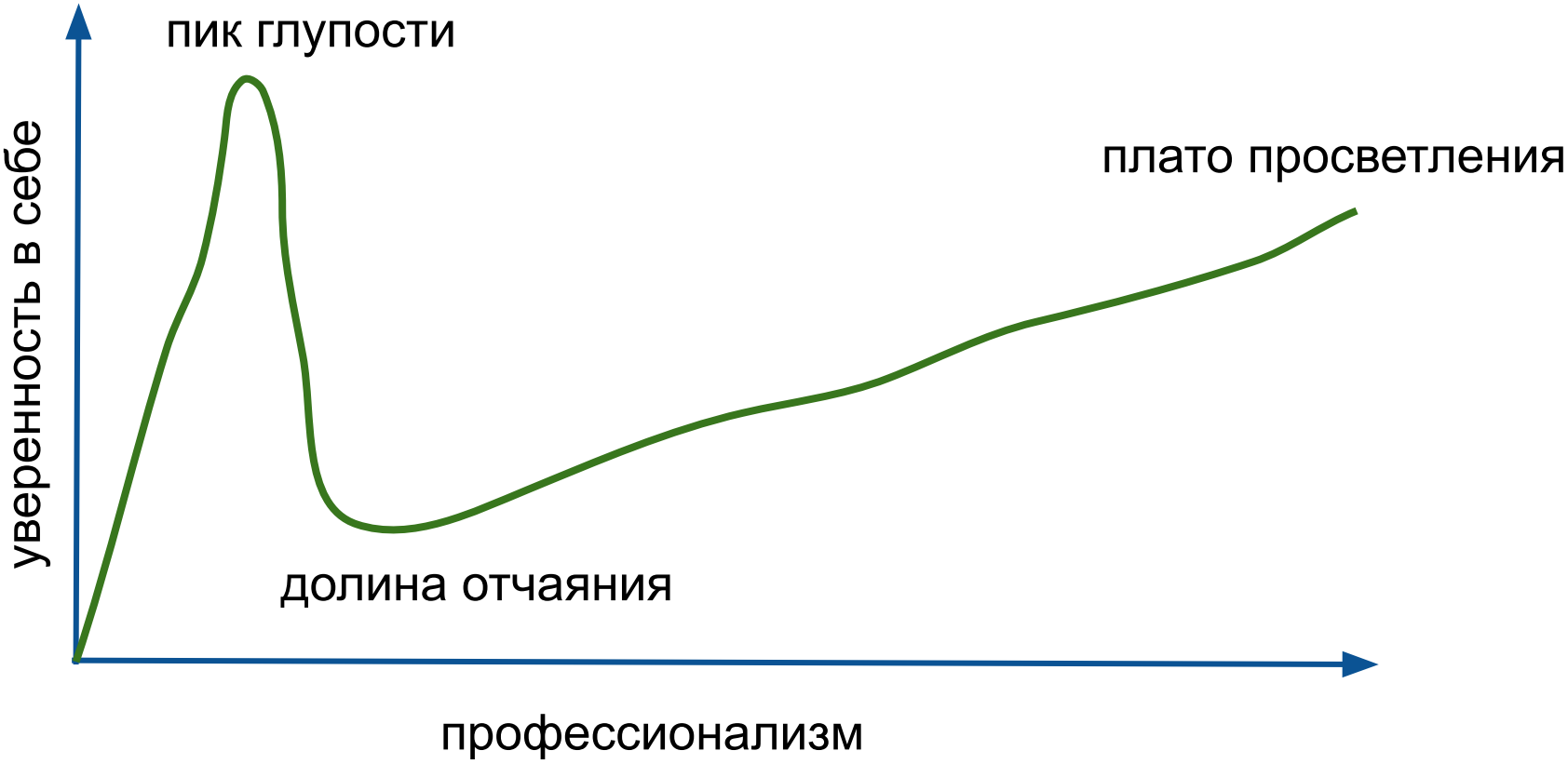
1. Вызов `getchar()` для чтения символа

2. Проверка на EOF

3. Если получен EOF, то вызывает `calloc(1, 1)` и возвращает результат

4. Иначе вызывает `calloc(2, 1)`, записывает в 0-й элемент полученный символ и возвращает результат

Эффект Даннинга-Крюгера



Подытожим

Пример к лекции

https://sourcecraft.dev/sshambir-public/pstigercpp

Ветки:

- `01_lexemes` — лексический анализ
- `02_input_output` — ввод-вывод и завершение программы

Конец!