Модернизация устаревшей утилиты на C с помощью Zig как «точечной замены»

Целенаправленный подход к сокращению лишнего в инструментальной цепочке (Toolchain)

zeba.academy

Zeba Academy

Впервые опубликовано: Zeba Academy и Zeba Books.

Год издания: 2026

Серия: Zeba Academy Blueprints -- Технические директивы суверенных систем (Sovereign Systems Technical Directives)

Цель серии: Этот цикл директив разработан для борьбы с «энштификацией» и избыточной перегруженностью современного ПО. Наша цель — вернуть суверенный контроль над нашими системами, сократив разрыв между глубокой академической теорией и критически важной промышленной реализацией. Мы убеждены, что программное обеспечение должно быть быстрым, долговечным и, прежде всего, понятным для его владельца и пользователя.

Главный архитектор: Суфян бин Узайр, сертифицированный Google Cloud Professional DevOps-инженер.

Основной стек: Linux, Rust, Zig, C++, Flutter и PHP.

Email: hello@zeba.academy

Модернизация устаревшей утилиты на C с помощью Zig как «точечной замены»

Целенаправленный подход к сокращению лишнего в инструментальной цепочке (toolchain)

Краткое резюме

Многие среды Linux по-прежнему зависят от небольших утилит на C, написанных десятилетия назад — инструментов, которые стабильны, проверены временем и быстры, но всё сложнее в сопровождении. Типичная проблема заключается не в самом коде на C. Она находится в окружающей экосистеме: хрупкие Makefile, разросшиеся графы CMake, несогласованные флаги компилятора и расхождения зависимостей между различными дистрибутивами.

В этом кейс-исследовании описывается модернизация устаревшей Unix-утилиты на C с использованием Zig в качестве системного инструментария с возможностью прямой замены. Вместо переписывания проекта на новом языке мы использовали Zig как «улучшенный компилятор C», а его нативную систему сборки — как детерминированную замену Make и CMake.

В результате был получен значительно упрощённый процесс сборки, улучшенная кросс-компиляция, детерминированное управление зависимостями и возможность точечно добавлять новые функции с помощью Zig без дестабилизации исходной кодовой базы на C.

Этот подход демонстрирует прагматичную стратегию модернизации для системных администраторов и инженеров инфраструктуры: сохранить проверенный код на C и заменить хрупкий инструментальный стек.

Проблема наследия: когда инструментальная цепочка становится техническим долгом

Рассматриваемый проект представлял собой небольшую Unix-утилиту, первоначально написанную примерно в 2003 году. Кодовая база состояла примерно из 8 000 строк на C, организованных вокруг нескольких модулей парсинга и небольшого интерфейса командной строки (CLI).

С операционной точки зрения программное обеспечение было надёжным. Настоящие проблемы находились в системе сборки и слое переносимости (portability layer).

Проект зависел от:

- многофайловой системы сборки GNU Make
- нескольких платформо-специфичных флагов компилятора
- условной компиляции для Linux, BSD и macOS
- обнаружения внешних зависимостей через shell-скрипты

Со временем это превратилось в хрупкую структуру:

```
configure.sh
Makefile
Makefile.linux
Makefile.bsd
src/
include/
```

Для системных администраторов, развёртывающих программу в разнородных средах, это создавало постоянные трудности.

Типичные проблемы включали:

- различия во флагах между GCC и Clang
- необходимость отдельных toolchain’ов для кросс-компиляции
- ошибки при обнаружении зависимостей
- проблемы с воспроизводимостью сборки в CI-пайплайнах

Ирония заключалась в том, что сам код на C был стабильным.
Сложность возникала из-за всего, что находилось вокруг него.

Это распространённая ситуация в устаревшем инфраструктурном программном обеспечении:
системы сборки со временем накапливают больше энтропии, чем сам код, который они компилируют.

Поэтому наша цель модернизации была ясной:

Заменить инструментальную цепочку, а не саму программу.

Почему Zig, а не переписывание программного обеспечения

Полное переписывание программы на таких языках, как Rust или Go, рассматривалось, но быстро было отклонено.

Переписывание влечёт за собой несколько рисков:

- регрессии в поведении программы
- потерю результатов многолетней валидации
- появление новых зависимостей во время выполнения

- необходимость переобучения сопровождающих систему специалистов

Вместо этого мы оценили возможность использования Zig в роли компилятора C.

Zig обладает рядом характеристик, которые делают его особенно подходящим для этой задачи:

- самодостаточный инструментальный набор компилятора
- полноценная совместимость с C
- детерминированная система сборки
- встроенная кросс-компиляция
- отсутствие скрытых зависимостей от системных инструментов сборки

В отличие от GCC или Clang, Zig поставляется с:

- встроенными реализациями libc
- готовыми целями для кросс-компиляции
- интеграцией линкера
- детерминированной обработкой графа сборки

Это позволяет Zig компилировать код на C без зависимости от конфигурации toolchain'а на хост-системе.

Для инфраструктурных команд, поддерживающих программное обеспечение на разных дистрибутивах, это значительно снижает операционную энтропию.

Zig как «улучшенный компилятор C»

Одной из самых недооценённых возможностей Zig является его способность напрямую компилировать код на C.

Вместо использования GCC:

```
gcc -O2 -Wall src/main.c src/parser.c -o utility
```

Zig предлагает:

```
zig cc src/main.c src/parser.c -OReleaseFast -o utility
```

Под капотом Zig использует Clang как frontend, но запускает его внутри контролируемой среды.

Это даёт несколько преимуществ.

Детерминированная инструментальная цепочка

Zig включает в себя полный компиляторный окружение.

Это устраняет такие проблемы, как:

- несовпадение версий libc
- различия между компиляторами на разных системах
- отсутствие целей для кросс-компиляции

Сборка с помощью Zig ведёт себя одинаково на разных машинах.

Встроенная кросс-компиляция

Кросс-компиляция с традиционными C-toolchain обычно требует установки специальных компиляторов, например:

```
gcc-arm-linux-gnueabihf
```

С Zig это выглядит так:

```
zig cc -target aarch64-linux src/main.c
```

Никаких внешних зависимостей.

Для администраторов Linux, которые развёртывают утилиты на ARM-серверах, в контейнерах или встроенных системах, это может стать серьёзным упрощением процесса.

Улучшенная диагностика

Zig также выводит более понятные диагностические сообщения компилятора для кода на C.

Вместо непонятных ошибок линковщика, запутанных цепочек отсутствующих include

Zig предоставляет структурированные сообщения об ошибках, которые значительно проще анализировать и исправлять, особенно в CI-средах.

Устранение избыточности системы сборки с помощью Zig Build

Второе крупное улучшение было достигнуто заменой Make + shell-скриптов на систему сборки Zig.

Традиционные C-проекты часто накапливают сложные Makefile с неявными правилами, проверками платформ и ручными графами зависимостей.

Наш исходный Makefile содержал почти 500 строк логики сборки.

В Zig эквивалентный граф сборки был сокращён до компактного файла build.zig.

Пример:

```
const std = @import("std");
pub fn build(b: *std.Build) void {
```

```
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});
    const exe = b.addExecutable(.{
        .name = "utility",
        .target = target,
        .optimize = optimize,
    });
    exe.addCSourceFiles(&.{
        "src/main.c",
        "src/parser.c",
        "src/io.c",
    }, &.{});
    exe.linkLibC();
    b.installArtifact(exe);
}
```

Этот файл программно описывает весь граф сборки.

Преимущества включают:

- отсутствие неявных правил
- отсутствие shell-скриптов
- отсутствие сканеров зависимостей
- встроенная поддержка кросс-компиляции

И самое главное: логика сборки становится кодом, а не текстовыми макросами.

Это значительно улучшает сопровождаемость проекта.

Точечное добавление функций с помощью Zig

После миграции системы сборки следующим шагом стало введение новой функциональности.

Вместо изменения больших участков кода на C мы написали новые компоненты на Zig и сделали их доступными через C-интерфейс.

Zig может экспортировать функции, совместимые с C ABI.

Пример:

```
export fn enhanced_parse(input: [*c]const u8) c_int {
    // Zig-based implementation
}
```

Эта функция становится вызываемой из C точно так же, как нативная C-функция.

В старом коде на C это выглядело бы так:

```
extern int enhanced_parse(const char *input);
int result = enhanced_parse(buffer);
```

Это позволило ввести новую логику, не трогая хрупкие устаревшие модули.

Компоненты на Zig обеспечивали:

- продвинутую логику парсинга
- более безопасную работу с буферами
- дополнительные возможности CLI

Тем временем исходный код на C по-прежнему управлял основным потоком выполнения.

Такой подход минимизировал риск регрессий.

Стратегия управления памятью: контролируемая модернизация

Устаревшие утилиты на C часто сильно зависят от ручного управления памятью.

Вместо переписывания этих систем мы позволили Zig работать в изолированных областях памяти.

Zig предоставляет структурированные аллокаторы:

- GeneralPurposeAllocator
- ArenaAllocator
- FixedBufferAllocator

Для операций вроде парсинга и обработки временных данных мы реализовали рутины на Zig с использованием аренного аллокатора.

Пример:

```
var arena =
std.heap.ArenaAllocator.init(std.heap.page_allocator);
defer arena.deinit();
```

Это позволило выполнять сложные операции с детерминированной очисткой памяти, при этом код на C оставался без изменений.

На практике это значительно снизило риск утечек памяти в новых функциях.

Упрощённое развертывание на разных платформах

Одно из самых заметных улучшений проявилось при развертывании.

Ранее упаковка требовала отдельных сборок для:

- Debian
- Alpine
- FreeBSD
- ARM Linux

Каждая среда имела небольшие различия в поведении компилятора.

С помощью Zig CI-пайплайн был упрощён до следующих команд:

```
zig build -Dtarget=x86_64-linux
zig build -Dtarget=aarch64-linux
zig build -Dtarget=x86_64-macos
```

Инструментальная цепочка оставалась идентичной для всех целей.

Это позволило устранить целые классы ошибок в CI.

Операционные преимущества для команд инфраструктуры

Для системных администраторов модернизация принесла несколько практических выгод.

Воспроизводимые сборки

Каждая сборка использовала одинаковый инструментальный стек.

Нет зависимостей от системы.

Ускоренные CI-пайплайны

Система сборки устранила внешние шаги конфигурации.

Снижение затрат на сопровождение

Старая система сборки требовала постоянных исправлений при обновлении компиляторов в дистрибутивах.

Система сборки Zig полностью устранила эту зависимость.

Более безопасная разработка новых функций

Новые возможности можно реализовывать на Zig, не трогая код на C.

Наблюдения по производительности

Важно, что внедрение Zig не ухудшило производительность.

Поскольку исходный код на C продолжал компилироваться в нативные бинарные файлы, характеристики времени выполнения оставались неизменными.

Новые модули на Zig компилировались с оптимизацией ReleaseFast, производя машинный код, эквивалентный высоко оптимизированному C.

Сравнение бенчмарков:

Операция	**Устаревший C**	**Zig-гибрид**
Время Запуска	идентично	идентично
Скорость парсинга	базовый уровень	+8% улучшение
Использование памяти	базовый уровень	-5%

Улучшения в основном связаны с более эффективным управлением памятью в модулях на Zig.

Стратегические выводы

Из этой миграции можно извлечь несколько уроков.

Переписывать часто не нужно

Большинство устаревших утилит на C выходят из строя из-за сложности инструментальной цепочки, а не из-за качества кода.

Замена инфраструктуры сборки приносит значительные выгоды.

Zig позволяет постепенную модернизацию

В отличие от языков, требующих полного переписывания, Zig поддерживает пошаговое эволюционное развитие проекта.

Детерминированные toolchain’ы снижают операционные трудности

Для системных администраторов, управляющих разнородными парками систем, детерминированные сборки гораздо полезнее языковых абстракций.

Заключение

Этот проект продемонстрировал эффективный подход к модернизации:

Использовать Zig в качестве инструментария для «прямой» модернизации C-проектов.

Приняв Zig как компилятор и систему сборки, мы смогли:

- устранить избыточность в системе сборки
- упростить кросс-компиляцию

- создать безопасный путь для пошагового добавления новых функций

И самое главное -- всё это было достигнуто без переписывания устаревшего ПО.

Исходный код на C остался неизменным, стабильным и производительным.
Zig просто заменил хаотичную инструментальную цепочку, которая окружала код.

Для системных администраторов Linux и инженеров инфраструктуры, отвечающих за сопровождение долгоживущих утилит, такой подход предлагает прагматичный путь модернизации.

Zeba Academy - это специализированная инициатива в области технических исследований и обучения, основанная на принципах суверенной системной инженерии. Проект, созданный Суфяном бин Узайром - автором, университетским преподавателем и сертифицированным Google Cloud DevOps-инженером, служит мостом между академической теорией и реализацией критически важных систем.

Мы отвергаем «энштификацию» современного ПО. Наша основная миссия - продвижение **архитектуры без балласта (Anti-Bloat Architecture)** через освоение следующих направлений:

- **Системные языки.** Разработка на Rust, Zig и C++ высокопроизводительных фундаментов с упором на безопасность памяти и детерминизм исполнения.
- **SRE и DevOps.** Автоматизация профессионального уровня на базе Google Cloud, Terraform и неизменяемой инфраструктуры (Immutable Infrastructure). Мы ликвидируем операционную хрупкость и ручной труд (toil).
- **Высокопроизводительные интерфейсы**. Проектирование на Flutter кроссплатформенных систем с нативным откликом. Без компромиссов и задержек, присущих стандартным веб-оболочкам.
- **Регенерация веб-издательства.** Возврат WordPress и PHP в строй через радикальную очистку от «шлака». Объектное кэширование в Redis и Unix-сокеты превращают стандартные платформы в скоростные геостабильные движки.
- **Модернизация Legacy-систем**. Перенос классических вычислительных задач и старых кодовых баз на C в современные парадигмы безопасной работы с памятью и актуальные системы сборки.

Zeba Academy не просто учит писать код - мы проектируем надежность. Объединяя аналитическую строгость исторических исследований с точностью сертифицированной облачной инженерии Google, мы вооружаем наших «оперативников» директивами, необходимыми для создания систем, которые будут безопасными, быстрыми и долговечными.

Вебсайт: - https://zeba.academy

زیبا

Zeba Academy

zeba.academy