Тулчейн RISC‑V и компиляторные оптимизации

2023-12-11

Данный материал является переводом и адаптацией учебного курса RISC‑V Toolchain and Compiler Optimization Techniques (LFD113x), The Linux Foundation, <https://training.linuxfoundation.org/training/risc-v-toolchain-and-compiler-optimization-techniques-lfd113x/>, распространяемого под лицензией [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/). Перевод и адаптация выполнены Кириленко Яковом Александровичем по заказу [Альянса RISC‑V](https://riscv-alliance.ru/), допускается к использованию под лицензией [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/deed.ru).

Обращаем внимание читателя, что данный текст не является строгим академическим изложением, а представляет собой перевод с некоторым сглаживанием и уточнением терминологии, а также добавлением русскоязычных справочных материалов. Перед прочтением курса рекомендуем ознакомиться с используемыми в нём терминами и их переводами, список которых можно найти в разделе [glossary\_title](#section-terms).

# Системы компиляции

Технологии компиляции стали важны как никогда. Как небольшие, так и крупные технологические компании нанимают разработчиков компиляторов. С ростом использования IoT-устройств и машинного обучения потребность в таких инженерах за последние несколько лет резко возросла.

У большинства крупных компаний есть разработчики систем компиляции[[1]](#footnote-24), работающие над компиляторами с открытым или закрытым исходным кодом. Некоторые инженеры улучшают сами компиляторы, некоторые — работают над оптимизацией приложений, потребляющих много ресурсов.

После ознакомления с данной главой вы будете:

* разбираться в системах компиляции и инструментах, используемых при создании приложений;
* понимать, как использовать популярные инструменты такие, как GCC и LLVM, для создания приложений, в частности приложений для RISC‑V;
* разбираться в структуре систем компиляции и знать, к каким ресурсам обращаться при возникновении вопросов;
* понимать концепцию кросс-компиляции и sysroot.

## Введение в компиляторы

Компилятор — система, преобразующая язык программы с одного на другой. В рамках курса компилятором будем называть программу, которая преобразует языки высокого уровня, такие как C или C++, в язык низкого уровня: язык ассемблера или исполняемый формат. Для этого обычно используются довольно продвинутые компиляторы с открытым исходным кодом, например, GCC или Clang.

Вкратце, они:

* Компилируют несколько языков высокого уровня: C, C++, Fortran, Objective C и другие.
* Предназначены для разных архитектур: ARM, AArch64, MIPS, RISC‑V, WebAssembly, x86-64 и другие.
* Оптимизируют скорость работы программы: разворачивают циклы, заменяют вызовы функций их определениями, векторизуют циклы и так далее.
* Обеспечивают статический анализ, предупреждения о возможных ошибках и др.
* Предоставляют программный интерфейс для других инструментов интроспекции и преобразования исходного кода.
* Предоставляют возможности инструментирования исходного кода для анализа производительности и интроспекции.

## Что такое системы компиляции?

Какие зависимости необходимы для компиляции простой программы вроде hello-world? Даже для такой программы нужны заголовочные файлы и библиотеки. Заголовочный файл, например, iostream, используется во время компиляции для поиска объявления функций, отсутствующих в самой программе (например, std::cout), а библиотеки — их определений (например, std::operator<<) во время компоновки. Результат компиляции — исполняемый файл, запускающийся на машине.

## Процесс компиляции

Если для компиляции программы на языке C++ используется компилятор вроде g++, процесс компиляции включает в себя несколько этапов, в зависимости от того, какой результат требуется получить. Для просмотра этапов компиляции нужно передать компилятору флаг -v. Для понимания процесса компиляции рассмотрим небольшую программу hello-world, представленную ниже:

#include<iostream>  
int main() {  
 std::cout << "Hello world\n";  
 return 0;  
}

Давайте рассмотрим подробный вывод вызова компилятора g++. Хотя подробный вызов выводит много информации, интересующими нас строками являются вызов компилятора, вызов ассемблера и вызов компоновщика. Мы только что сказали, что g++ hello.cpp — это вызов компилятора. Но это так лишь отчасти, поскольку g++ — это не компилятор, а драйвер компилятора (компилятор-драйвер). Драйвер компилятора — это программа, которая вызывает различные инструменты в цепочке инструментов компилятора для перевода исходного кода на целевой язык.

Компилятором в данном случае является cc1plus, вызов приведен ниже:

g++ hello.cpp -v

Using built-in specs.  
COLLECT\_GCC=g++  
COLLECT\_LTO\_WRAPPER=/usr/lib/gcc/x86\_64-linux-gnu/11/lto-wrapper  
OFFLOAD\_TARGET\_NAMES=nvptx-none:amdgcn-amdhsa  
OFFLOAD\_TARGET\_DEFAULT=1  
Target: x86\_64-linux-gnu  
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 11.3.0-1ubuntu1~22.04' --with-bugurl=file:///usr/share/doc/gcc-11/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,m2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-11 --program-prefix=x86\_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --enable-libphobos-checking=release --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --enable-cet --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-11-xKiWfi/gcc-11-11.3.0/debian/tmp-nvptx/usr,amdgcn-amdhsa=/build/gcc-11-xKiWfi/gcc-11-11.3.0/debian/tmp-gcn/usr --without-cuda-driver --enable-checking=release --build=x86\_64-linux-gnu --host=x86\_64-linux-gnu --target=x86\_64-linux-gnu --with-build-config=bootstrap-lto-lean --enable-link-serialization=2  
Thread model: posix  
Supported LTO compression algorithms: zlib zstd  
gcc version 11.3.0 (Ubuntu 11.3.0-1ubuntu1~22.04)  
COLLECT\_GCC\_OPTIONS='-v' '-shared-libgcc' '-mtune=generic' '-march=x86-64' '-dumpdir' 'a-'  
 /usr/lib/gcc/x86\_64-linux-gnu/11/cc1plus -quiet -v -imultiarch x86\_64-linux-gnu -D\_GNU\_SOURCE hello.cpp -quiet -dumpdir a- -dumpbase hello.cpp -dumpbase-ext .cpp -mtune=generic -march=x86-64 -version -fasynchronous-unwind-tables -fstack-protector-strong -Wformat -Wformat-security -fstack-clash-protection -fcf-protection -o /tmp/ccwbNcfY.s  
GNU C++17 (Ubuntu 11.3.0-1ubuntu1~22.04) version 11.3.0 (x86\_64-linux-gnu)  
 compiled by GNU C version 11.3.0, GMP version 6.2.1, MPFR version 4.1.0, MPC version 1.2.1, isl version isl-0.24-GMP  
  
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072  
ignoring duplicate directory "/usr/include/x86\_64-linux-gnu/c++/11"  
ignoring nonexistent directory "/usr/local/include/x86\_64-linux-gnu"  
ignoring nonexistent directory "/usr/lib/gcc/x86\_64-linux-gnu/11/include-fixed"  
ignoring nonexistent directory "/usr/lib/gcc/x86\_64-linux-gnu/11/../../../../x86\_64-linux-gnu/include"  
#include "..." search starts here:  
#include <...> search starts here:  
 /usr/include/c++/11  
 /usr/include/x86\_64-linux-gnu/c++/11  
 /usr/include/c++/11/backward  
 /usr/lib/gcc/x86\_64-linux-gnu/11/include  
 /usr/local/include  
 /usr/include/x86\_64-linux-gnu  
 /usr/include  
End of search list.  
GNU C++17 (Ubuntu 11.3.0-1ubuntu1~22.04) version 11.3.0 (x86\_64-linux-gnu)  
 compiled by GNU C version 11.3.0, GMP version 6.2.1, MPFR version 4.1.0, MPC version 1.2.1, isl version isl-0.24-GMP  
  
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072  
Compiler executable checksum: 449548cbb29044828dc7ea158b577b98  
COLLECT\_GCC\_OPTIONS='-v' '-shared-libgcc' '-mtune=generic' '-march=x86-64' '-dumpdir' 'a-'  
 as -v --64 -o /tmp/ccAOJYkw.o /tmp/ccwbNcfY.s  
GNU assembler version 2.38 (x86\_64-linux-gnu) using BFD version (GNU Binutils for Ubuntu) 2.38  
COMPILER\_PATH=/usr/lib/gcc/x86\_64-linux-gnu/11/:/usr/lib/gcc/x86\_64-linux-gnu/11/:/usr/lib/gcc/x86\_64-linux-gnu/:/usr/lib/gcc/x86\_64-linux-gnu/11/:/usr/lib/gcc/x86\_64-linux-gnu/  
LIBRARY\_PATH=/usr/lib/gcc/x86\_64-linux-gnu/11/:/usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu/:/usr/lib/gcc/x86\_64-linux-gnu/11/../../../../lib/:/lib/x86\_64-linux-gnu/:/lib/../lib/:/usr/lib/x86\_64-linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86\_64-linux-gnu/11/../../../:/lib/:/usr/lib/  
COLLECT\_GCC\_OPTIONS='-v' '-shared-libgcc' '-mtune=generic' '-march=x86-64' '-dumpdir' 'a.'  
 /usr/lib/gcc/x86\_64-linux-gnu/11/collect2 -plugin /usr/lib/gcc/x86\_64-linux-gnu/11/liblto\_plugin.so -plugin-opt=/usr/lib/gcc/x86\_64-linux-gnu/11/lto-wrapper -plugin-opt=-fresolution=/tmp/cclx7mGg.res -plugin-opt=-pass-through=-lgcc\_s -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc\_s -plugin-opt=-pass-through=-lgcc --build-id --eh-frame-hdr -m elf\_x86\_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro /usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu/crti.o /usr/lib/gcc/x86\_64-linux-gnu/11/crtbeginS.o -L/usr/lib/gcc/x86\_64-linux-gnu/11 -L/usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu -L/usr/lib/gcc/x86\_64-linux-gnu/11/../../../../lib -L/lib/x86\_64-linux-gnu -L/lib/../lib -L/usr/lib/x86\_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86\_64-linux-gnu/11/../../.. /tmp/ccAOJYkw.o -lstdc++ -lm -lgcc\_s -lgcc -lc -lgcc\_s -lgcc /usr/lib/gcc/x86\_64-linux-gnu/11/crtendS.o /usr/lib/gcc/x86\_64-linux-gnu/11/../../../x86\_64-linux-gnu/crtn.o  
COLLECT\_GCC\_OPTIONS='-v' '-shared-libgcc' '-mtune=generic' '-march=x86-64' '-dumpdir' 'a.'

Как видно из команды, cc1plus компилирует hello.cpp и выводит ассемблерный код в файл /tmp/ccwbNcfY.s. Во время компиляции cc1plus должен найти заголовочный файл iostream, который находится в /usr/include/c++/11.

Далее идет вызов ассемблера. Он считывает вывод компилятора (т. е. /tmp/ccwbNcfY.s) и выводит объектный файл /tmp/ccAOJYkw.o. Ассемблер не имеет никаких зависимостей:

as -v --64 -o /tmp/ccAOJYkw.o /tmp/ccwbNcfY.s

И, наконец, у нас есть вызов компоновщика. Компоновщик collect2 считывает выходные данные ассемблера /tmp/ccAOJYkw.o, объектный файл и выводит исполняемый файл. Компоновщик имеет много зависимостей. Наиболее интересными из них являются файлы поддержки времени выполнения, а именно crt1.o, crti.o, crtendS.o, crtn.o, и стандартные библиотеки libc, libgcc, libgcc\_s, libm и т. д. Посмотрите, сможете ли вы определить, как эти зависимости передаются компоновщику. Теперь компоновщику нужно знать, где находятся эти файлы на самом деле, драйвер компилятора g++ должен знать, где находятся эти файлы, чтобы он мог вызвать компоновщик с соответствующими библиотеками (см. флаги, начинающиеся с -l) и соответствующими путями (см. флаги, начинающиеся с -L).

Итак, система компиляции — это набор инструментов, вспомогательных библиотек и заголовочных файлов, которые помогают собрать программу из исходного кода в исполняемый файл, который может выполняться на компьютере. Обратите внимание, что системы компиляции необходимы для создания исполняемых файлов, но только их недостаточно. Чего не хватает в системе компиляции, чтобы иметь «всё», что необходимо для создания исполняемых программ, так это sysroot.

## Sysroot

Любой компилятор должен «знать», где находятся стандартные заголовочные файлы, стандартные библиотеки и среда выполнения. Все они упакованы вместе для каждой платформы (например, arm64, x86) в каталоге с именем sysroot. Когда мы компилируем программу, нам нужно передать путь к sysroot, чтобы компилятор знал, где искать стандартные заголовочные файлы во время компиляции и где искать общие библиотеки (libc, libstdc++ и т. д.) во время компоновки.

Обычно, когда мы компилируем программу для той же машины, компилятор использует стандартные заголовочные файлы, доступные в /usr/include, и библиотеки из /usr/lib. Эти пути встроены в исходный код самого компилятора, так что нам никогда не придётся об этом думать. Однако при создании собственного компилятора или при кросс-компиляции программ мы сообщаем компилятору, где находится sysroot, передавая флаг (например, gcc --sysroot="/path/to/arm64/sysroot/usr" hello.cpp). Чаще всего предварительно упакованные кросс-компиляторы поставляются со сценарием/двоичным файлом, в который встроен путь sysroot (например, aarch64-linux-gnu-gcc) пакет [g++-10-aarch64-linux-gnu (10.4.0-4ubuntu1~22.04cross1 и другие)](https://packages.ubuntu.com/ru/kinetic/g++-10-aarch64-linux-gnu).

## Система компиляции

Помимо sysroot, система компиляции содержит различные двоичные файлы, помогающие в процессе компиляции. В некоторых случаях сам компилятор входит в систему. Ниже приведен список элементов, упакованных с системой компиляции:

* binutils (ассемблер, компоновщик и т.д.);
* различные компиляторы (gcc, g++ и т.д.);
* C-библиотеки (glibc, uClibc и т.д.);
* библиотеки поддержки времени выполнения (crtbegin.o, crtend.o и т. д.);
* отладчик (gdb);
* стандартные заголовочные файлы C/C++ (iostream, stdio.h и т.д.);
* стандартные библиотеки (libstdc++, libm, libgcc, libunwind и т.д.);
* заголовочные файлы компилятора (stdint.h, stdc-predef.h);
* библиотеки поддержки времени выполнения для инструментов отладки (libasan, libubsan и т. д.).

**Примечание:** в предоставляемой системе компиляции могут присутствовать не все из них, в зависимости от поставщика. С более подробной информацией можно ознакомиться здесь:

* [Toolchains](https://elinux.org/Toolchains)
* [How Initialization Functions Are Handled (C runtime)](https://gcc.gnu.org/onlinedocs/gccint/Initialization.html)

### Полезные ресурсы

* [Константин Владимиров — Специальный выпуск: лекции по тулчейну, часть 1, фронтенд](https://youtu.be/335ylTUlyng)
* [Константин Владимиров — Специальный выпуск: лекции по тулчейну, часть 2, миддленд и ассемблер](https://youtu.be/04Gt0HOatUw)
* [Константин Владимиров — Специальный выпуск: лекции по тулчейну, часть 3, линкеры](https://youtu.be/lUvcvmG6kIw)

## Кросс-компиляция

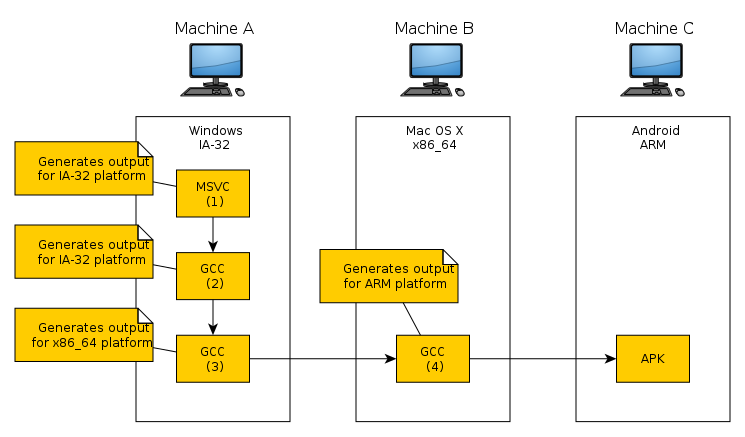
Чтобы понять концепцию кросс-компиляции, давайте вернемся к определению компилятора. Компилятор — это программа, которая преобразует программу с одного языка на другой. Но слово «компилятор» часто используется для обозначения программы, которая переводит программу на машинный язык для создания исполняемого файла, работающего на вычислительном устройстве. Обычно компилятор используется для генерации машинного кода для той же машины, на которой работает сам компилятор. Под одной и той же машиной мы подразумеваем одну и ту же архитектуру. Например, компилятор, работающий на машине с linux-x64, компилирует программу на C++ и генерирует машинный код для той же машины с linux-x64. Эта программа может работать на всех машинах linux-x64, если предоставляется аналогичная среда.

Однако бывают ситуации, когда мы хотим сгенерировать двоичные файлы для машин другого типа, а не того, на котором запускается компилятор. Например, если целевая машина недостаточно мощная. Это часто имеет место при создании двоичных файлов для встраиваемых устройств, мобильных приложений и т. д. Кросс-компилятор создает двоичные файлы, которые будут работать на другом компьютере (целевом компьютере), а не на том, на котором работает сам компилятор (хост-компьютер). Это немного более сложный процесс, так как он требует присутствия всех зависимостей целевой машины на хост-машине.

Например, при компиляции простой программы hello-world для хост-компьютера используется заголовочный файл stdio.h на месте, подобном /usr/include/stdio.h. Для создания кросс-компилируемой программы hello-world файл stdio.h будет находиться в другом sysroot. Итак, вызов компилятора может выглядеть так:

gcc --sysroot=/path/to/aarch64/sysroot -march=armv8-a hello.c

Еще более запутанная система — [канадская сборка](https://en.wikipedia.org/wiki/Cross_compiler#Canadian_Cross)[[2]](#footnote-38). В ней участвуют два кросс-компилятора. В этой настройке есть три машины A, B и C. Кросс-компилятор в A (CA) сгенерирует другой кросс-компилятор (CB), который будет работать на B. CB сгенерирует код для машины C.



Схематичный пример канадской сборки

## Введение в систему компиляции GCC

Популярный компилятор gcc поставляется вместе с тулчейном gcc. Его можно загрузить с [зеркальных сайтов GCC](https://gcc.gnu.org/mirrors.html). Чтобы загрузить предварительно скомпилированные двоичные файлы, готовые к использованию, перейдите на страницу [Установка GCC: двоичные файлы](https://gcc.gnu.org/install/binaries.html).

Тулчейн содержит следующие каталоги верхнего уровня:

* bin
* include
* lib
* libexec
* share

Папка bin содержит все исполняемые двоичные файлы, такие как компилятор C (gcc), компилятор C++ (g++), компилятор Fortran (gfortran) и компилятор D (gdc). В зависимости от дистрибутива он может содержать компилятор Go (gccgo). Он также содержит набор других полезных инструментов, таких как gcov, lto-dump и т.д.

Папка include содержит набор заголовочных файлов, которые включаются во время компиляции. Например, заголовочные файлы C++, такие как iostream и т.д. Обратите внимание, что заголовочные файлы С, такие как stdio.h, не поставляются с тулчейном, потому что они являются частью sysroot.

Папка lib содержит такие библиотеки, как libstdc++, libatomic и т. д. Эти библиотеки могут использоваться компилятором gcc в процессе компиляции или могут использоваться как повторно используемый набор библиотек.

Папка libexec содержит двоичные файлы, которые вызываются программами-драйверами (gcc, g++, gdc). Например, gcc вызывает cc1 (компилятор C), collect2 (компоновщик), lto1 (оптимизатор времени компоновки) и т.д.

В папке share содержится документация, которая может быть установлена в виде man-страниц, а также неосновной набор скриптов.

### Полезные ресурсы

* [Онлайн документация GCC](https://gcc.gnu.org/onlinedocs/)
* [Ресурсный центр GCC](https://www.cse.iitb.ac.in/grc/)
* [Основные абстракции в GCC](https://www.cse.iitb.ac.in/grc/index.php?page=gcc-pldi14-tut)
* [Видеозаписи лекций по основным абстракциям в GCC -2012](https://www.cse.iitb.ac.in/grc/index.php?page=videos)
* [Лекция по внутреннему устройству компилятора в Индийском технологическом институте Бомбея [Часть 1](https://www.youtube.com/watch?v=IlovhbAI7Cw&list=PLy-CGmBdq2VGjl56cyaEjxcAMyAvUKbCz)]
* [GNU Tools Cauldron](https://www.youtube.com/channel/UCQ4JGczdlU3ofHWf3NuCX8g/featured)

### Список рассылок

* [Архив Gcc-help](https://gcc.gnu.org/pipermail/gcc-help/)

## Введение в систему компиляции LLVM

Проект LLVM — это коллекция модульных и переиспользуемых технологий компилятора и тулчейна. Предварительно собранные двоичные файлы системы компиляции LLVM можно загрузить со [страницы LLVM](https://releases.llvm.org/download.html). Тулчейн также можно собрать из исходных файлов, следуя инструкциям со страницы [документации](https://llvm.org/docs/). Тулчейн содержит следующие каталоги верхнего уровня:

* bin
* include
* lib
* libexec
* share

Папка bin содержит все исполняемые двоичные файлы, такие как компилятор clang и набор других полезных инструментов, таких как clang-rename, clang-refactor.

Папка include содержит набор заголовочных файлов, которые включаются во время компиляции. Например, заголовочные файлы C++, например, iostream. Обратите внимание, что заголовочные файлы C, такие как stdio.h, не поставляются с набором инструментов, потому что они являются частью sysroot. Он также содержит заголовочные файлы, которые используются при использовании библиотек llvm для сборки инструментов.

Папка lib содержит такие библиотеки, как libc`, `libcabi. Они могут использоваться компилятором clang в процессе компиляции или могут использоваться как повторно используемый набор библиотек.

Папка libexec содержит два скрипта Python, которые подходят только для использования статического анализатора clang.

Папка share содержит документацию, которую можно установить в виде справочных страниц, и необязательный набор скриптов.

### Полезные ресурсы

* [Документация по инфраструктуре компилятора LLVM](https://llvm.org/docs/)
* [Канал LLVM на YouTube](https://www.youtube.com/channel/UCv2_41bSAa5Y_8BacJUZfjQ)

### Список рассылок

* [Архив llvm-dev](https://lists.llvm.org/pipermail/llvm-dev/)

# Введение в компиляторные оптимизации

В этой главе вы узнаете о часто используемых компиляторных оптимизациях, которые используются для уменьшения размера кода и повышения производительности, а также о различных инструментах и методах профилирования.

Современные оптимизирующие компиляторы, такие как GCC и LLVM, предоставляют широкий спектр методов оптимизации и инструментирования. В то время как компиляторная оптимизация может повысить производительность или уменьшить размер кода, различные методы инструментирования могут помочь понять внутреннее устройство системы. Широко используемые уровни оптимизации достаточно хороши для большинства популярных аппаратных и программных экосистем. Однако требования современных систем, таких как приложения RISC‑V, работающих на устройствах IoT, недостаточно изучены в отрасли. В результате такие системы часто неэффективно оптимизируются с настройками компилятора по умолчанию. Понимание популярных методов оптимизации и инструментирования позволит рассмотреть альтернативы, при попытке повысить производительность, уменьшить размер кода или повысить безопасность приложения.

В этой главе вы узнаете, как использовать популярные методы компиляции для:

* улучшения производительности приложений;
* улучшения методов инструментирования;
* уменьшения размера кода;
* улучшения характеристик производительности некоторых встраиваемых приложений.

## Терминология

Прежде чем познакомиться с различными методами оптимизации, кратко ознакомимся с терминологией, которая будет часто использоваться в дальнейшем. Приведенные ниже термины предназначены только для помощи в понимании курса и не должны рассматриваться как строгие определения.

**Производительность**

Говоря о производительности приложения, мы обычно имеем в виду, сколько времени требуется для выполнения определенной задачи. Приложение должно выполнять задачи в разумные сроки, чтобы быть практически полезным. Во многих случаях мы хотим, чтобы приложения работали как можно быстрее. Есть разные способы повысить производительность приложений, один из них — использование компиляторных оптимизаций, мы обсудим в этой главе. Следует отметить, что не все части программы должны быть производительными, чтобы их можно было использовать на практике. Только определенные части, которые часто называют «узким местом» (ограничение, при котором теряется доля производительности или пропускной возможности системы), должны быть максимально производительными. Подробнее с темой производительности систем можно ознакомиться по ссылкам в справочном разделе.

**Компиляторные оптимизации**

Компиляторы выполняют ряд преобразований исходного кода. Хотя некоторые преобразования необходимы для генерации машинного кода, большинство преобразований выполняется для повышения производительности программ или уменьшения размера кода. Эти преобразования называются оптимизациями компилятора. В данной главе представлены оба типа оптимизации. Цель этой главы — дать учащимся возможность эффективно использовать оптимизации. Мы не обсуждаем, как эти оптимизации реализованы в компиляторах.

**Инструментирование**

Когда компилятор преобразует исходный код, он также может встраивать дополнительный код в программу. Эти преобразования называются инструментированием. Оно используется во многих случаях, одна из распространенных целей — сбор профиля времени выполнения программы. Чтобы собрать профиль времени выполнения, компилятор должен вставить счётчики в определенные части программы, и эти счётчики будут увеличиваться каждый раз, когда выполнение программы достигает места инструментирования. После завершения программы счётчики можно использовать для понимания профиля производительности. Самые ресурсоёмкие части программы наиболее интересны инженерам по производительности.

**Флаги компилятора**

Промышленные компиляторы, такие как GCC и LLVM, имеют сотни флагов, которые влияют на поведение компилятора. Существует множество флагов компилятора, и нет простого способа их классификации. Но для простоты мы попытаемся классифицировать флаги, чтобы облегчить понимание различных типов флагов:

* **Флаги оптимизации**
* Такие флаги, как -O2, -O3, -funroll-loops можно отнести к флагам оптимизации, поскольку они указывают компилятору, какие оптимизации следует выполнить.
* **Диагностические флаги**
* Такие флаги, как -Wall, -Werror, -Wnull-dereference влияют на диагностические выходные данные компилятора.
* **Настройка параметров**
* Флаги типа --param max-inline-insns-small=70 принимают разные значения, часто числовые, чтобы настроить, какая часть конкретной оптимизации будет выполнена.
* **Флаги инструментирования**
* Такие флаги, как -finstrument-function, -profile-generate, включают инструментирование. Инструментированный двоичный файл будет собирать профили времени выполнения, которые могут помочь с оптимизацией, обнаружением ошибок и т. д.
* **Флаги компоновщика**
* Такие флаги, как -lpthread, который используется компоновщиком для поиска определений символов, принятия решений по оптимизации и т. д.
* **Флаги, предоставляющие значение**
* Такие флаги, как -D, -fprofile-use, -stdlib=libstdc++, предоставляют компилятору дополнительные входные данные, которые могут помочь в оптимизации, диагностике, инструментировании и т. д.

## Оптимизация производительности

Компиляторы предлагают различные оптимизации для повышения производительности и/или уменьшения размера кода. Набор оптимизаций компилятора объединяется в зонтичных флагах компилятора, называемых «уровнями оптимизации». Ниже представлены уровни оптимизации, распространенные среди большинства компиляторов:

**-O0**

Это тривиальный случай, когда оптимизация компилятора не выполняется. Тем не менее оптимизация для конкретного языка в соответствии с требованиями стандарта по-прежнему выполняется. Например, вычисления во время компиляции, требуемые стандартом C++, по-прежнему выполняются. Этот уровень очень полезен для целей отладки в сочетании с флагом компилятора -g. Поскольку -O0 не выполняет оптимизацию, время компиляции является самым быстрым, что весьма полезно для итеративной разработки.

**-O1**

На этом уровне включается множество оптимизаций, повышающих производительность программы. Например, развертывание циклов, встраивание функций, планирование инструкций и т. д. Этот уровень оптимизации используется редко, поскольку сейчас доступны более агрессивные уровни оптимизации.

**-O2**

Это один из самых популярных уровней оптимизации. Он позволяет использовать все оптимизации -O1, а также более агрессивные оптимизации в распределении регистров, планировании инструкций, частичном устранении избыточности и т. д. Этот уровень используется при построении кода с преобладанием ветвлений, например, операционных систем.

**-O3**

Этот уровень включает в себя все возможности -O2, а также некоторые современные оптимизации, такие как векторизация. -O3 является фактическим уровнем оптимизации для максимизации производительности большинства приложений. -O3 также используется для тестов производительности, поскольку в нем присутствуют все «проверенные в боях» оптимизации компилятора.

**-Ofast**

Это просто -O3 с -ffast-math. Флаг -ffast-math указывает компилятору ослабить некоторые требования[[3]](#footnote-64) арифметики с плавающей запятой, такие как ассоциативность и коммутативность. Во многих приложениях ошибки, возникающие после ослабления этих требований, вполне допустимы за счёт более высокой производительности. Без -ffast-math многие циклы с операциями с плавающей точкой не могут быть векторизованы.

**-Os**

-Os оптимизирует размер кода. Таким образом, большинство оптимизаций, увеличивающих размер кода, будут менее агрессивными на этом уровне. Это популярная оптимизация среди встраиваемых систем и мобильных приложений, поскольку размер кода там является большой проблемой.

**-g**

Чтобы иметь возможность отлаживать приложение с аннотациями исходного кода, компилятор должен предоставить дополнительную информацию в двоичном файле. Флаг -g указывает компилятору сделать это. Без этого флага отладчик будет показывать только имена глобальных символов и дизассемблер, поскольку он не может связать строку исходного кода со сборкой.

**-finstrument-functions**

Этот флаг используется для инструментирования входа и выхода функций. Инструментирование позволяет получить представление о поведении программ. При использовании этого флага также необходимо определить две функции \_\_cyg\_profile\_func\_enter и \_\_cyg\_profile\_func\_exit, которые вызываются соответственно при входе и выходе из каждой вызываемой функции. Если есть функции, которые не должны быть инструментированы, к ним можно добавить \_\_attribute\_\_ ((no\_instrument\_function)).

**-fprofile-generate**, **-fprofile-arcs**, **-pg**

Эти флаги используются для инструментирования программ с целью сбора профилей времени выполнения различных точек программы. Это позволяет компилятору проводить оптимизацию с учетом профиля в последующих компиляциях. В зависимости от того, какие флаги вы используете, могут быть достигнуты различные типы инструментирования. Подробный обзор различных флагов приведен на странице руководства [gcc(1) - Linux manual page](https://man7.org/linux/man-pages/man1/gcc.1.html).

**-fstack-protector, -fstack-protector-all, -fstack-protector-strong**

Эти опции инструментируют уязвимые функции путем вставки защитных переменных в кадр стека. Перед возвратом функции проверяется, что защитная переменная не была перезаписана, что позволяет убедиться в том, что стек не был поврежден. Это тривиальный способ улучшить защиту от атаки на переполнение буфера. Однако это может увеличить размер кода приложения. В случае, если это создает накладные расходы, с этим флагом можно компилировать только критически важные для безопасности части приложения. Более подробную информацию об использовании этого флага можно найти [здесь](https://www.keil.com/support/man/docs/armclang_ref/armclang_ref_cjh1548250046139.htm).

## Profile-Guided Optimization

Зная частоту выполнения различных точек программы, компилятор может принимать более хорошие решения по оптимизации в процессе компиляции. Многие оптимизации компилятора основаны на жестко закодированных эвристиках и статическом анализе программы. Эти эвристики могут быть не идеальны для разных точек программы и часто приводят к потере производительности. Компилятор имеет возможность учитывать поведение программы во время выполнения, читая «файл покрытия». Файл покрытия — это, по сути, гистограмма частот выполнения различных точек программы. Для создания файла покрытия существует две методики, которые подробно описаны ниже.

### Использование инструментирования

Компиляторы могут вставлять «счётчики» в интересующие точки программы для сбора профилей исполнения. Код инструментируется путем передачи компилятору команды -fprofile-generate. Пример использования:

gcc -O2 -fprofile-generate=/path/to/outputfile test.c -o a.out

Благодаря инструментированию само приложение затем будет регистрировать события/счётчики, которые могут быть использованы компилятором во время следующей компиляции. После завершения работы программы в каталоге /path/to/outputfile/ будет создан файл с расширением .gcda. Затем перекомпиляция приложения с помощью -fprofile-use=/path/to/outputfile приведет к созданию оптимизированной программы.

gcc -O2 -fprofile-use=/path/to/outputfile test.c -o b.out

b.out оптимизируется с помощью собранной на первом шаге информации о профиле. Компилятор часто оптимизирует размещение кода, вставку функций и циклы с учетом информации профиля. Обычно при использовании PGO (Profile-Guided Optimization) наблюдается повышение производительности более чем на 10%.

### Использование семплирующих профилировщиков

Семплирующие профилировщики, такие как Linux perf, используют аппаратные счётчики для регистрации определенных событий во время выполнения программы. Программы можно профилировать как с самого начала, так и во время их выполнения. Это делает семплирующий профилировщик удобным для непрерывного профилирования. Накладные расходы таких профилировщиков довольно малы по сравнению с традиционным PGO, поэтому данный подход масштабируется на большое количество систем. Ниже приведен типичный сценарий использования:

perf record -b ./a.out  
create\_gcov --binary=./a.out --profile=perf.data --gcov=a.gcov  
-gcov\_version=1  
gcc -O3 -fauto-profile=a.gcov test.c -o b.out

b.out оптимизирован с использованием информации о профиле выборки. create\_gcov — это инструмент, который преобразует perf.data в файл покрытия в формате gcov. Инструмент [perf](https://perf.wiki.kernel.org/index.php/Main_Page) имеет множество опций для записи различных аппаратных событий. Следует отметить, что не все события поддерживаются всем оборудованием, и не все функциональные возможности Linux perf поддерживаются в RISC‑V.

### Соображения при использовании PGO

Хотя PGO на основе инструментирования в целом прост в использовании, существуют определенные недостатки:

* Инструментирование программы компилятором влияет на некоторые оптимизации.
* Инструментированная программа работает медленнее, поэтому ее нельзя развернуть на большом количестве систем. Из-за этого собранный профиль может быть не очень качественным.

Эти недостатки могут касаться не всех систем, поэтому всегда следует взвешенно оценивать, какие технологии PGO использовать. Качество оптимизации с помощью профиля зависит от тестовых векторов, по которым программа выполнялась при сборе профиля. Но даже при ограниченных тестовых сценариях в некоторых случаях все равно выгодно проводить PGO, например, последовательность запуска программы не сильно меняется даже при очень разных тестовых сценариях.

В дополнение к этому, может возникнуть проблема переобучения в некоторых точках программы, если охват во время сбора профиля был недостаточным. При переоснащении программа может плохо работать в определенных случаях. Чтобы преодолеть это, желательно выполнять непрерывное профилирование и компиляцию. Эта проблема менее актуальна для систем, где программы не часто меняются.

### Полезные материалы

* [Константин Владимиров — Специальный выпуск: лекции по тулчейну, часть 5, анализ кода и профилировка](https://youtu.be/azYgxwXxFJ8)

## Оптимизация объема кода

Размер кода встраиваемого ПО был проблемой в течение очень долгого времени. В то время как хранилища становятся все дешевле и меньше, разработчики находят творческие способы увеличить размер кода за счёт добавления функций или ненужной программной инженерии. Компиляторы прошли долгий путь в оптимизации приложений по размеру кода. В то время как большинство оптимизаций компиляторов были направлены на производительность приложений, в последние годы мы наблюдаем рост оптимизаций размера кода. В этом разделе мы познакомимся с широко используемыми методами уменьшения размера кода приложений. Этот раздел состоит из трех частей:

* *Методы измерения*: инструменты для измерения размера двоичного файла.
* *Оптимизация компилятора*: флаги компилятора, которые могут помочь уменьшить размер двоичных файлов приложений.
* *Оптимизация исходного кода*: методы разработки программного обеспечения для уменьшения размера двоичных файлов приложений.

### Измерение размера кода и различных секций

Существует три популярных инструмента для измерения размера кода двоичного файла.

1. **size**: [GNU Binutils](https://www.gnu.org/software/binutils/)
2. **strings**: [GNU Binutils](https://www.gnu.org/software/binutils/)
3. [Bloaty](https://github.com/google/bloaty)

#### Size

Утилита **size** может показать размер каждой секции двоичного файла.

size gcc/11/libstdc++.dylib

\_\_TEXT \_\_DATA \_\_OBJC others dec hex  
1703936 65536 0 1851392 3620864 374000

#### Strings

Показывает все строки в двоичном файле.

strings gcc/11/libstdc++.dylib | wc -l

2180

#### Bloaty

Этот инструмент может быть использован для более глубокого анализа двоичных файлов различных платформ. Он также сопоставляет исходным файлам их размеры в скомпилированном виде.

bloaty gcc/11/libstdc++.dylib

FILE SIZE VM SIZE  
-------------- --------------  
 29.1% 1.00Mi 29.0%. 1.00Mi \_\_TEXT,\_\_text  
 25.0% 882Ki 25.0% 882Ki String Table  
 16.6% 583Ki 16.5% 583Ki Symbol Table  
 12.3% 433Ki 12.2% 433Ki \_\_TEXT,\_\_eh\_frame  
 5.0% 176Ki 5.0% 176Ki Export Info  
 4.1% 146Ki 4.1% 146Ki \_\_TEXT,\_\_const  
 2.5% 87.8Ki 2.5% 87.8Ki Weak Binding Info  
 1.2% 41.6Ki 1.2% 41.6Ki \_\_DATA,\_\_gcc\_except\_tab  
 1.0% 36.9Ki 1.0% 36.9Ki \_\_DATA\_CONST,\_\_const  
 0.9% 33.3Ki 0.9% 33.3Ki \_\_TEXT,\_\_text\_cold  
 0.5% 16.1Ki 0.5% 16.1Ki [10 Others]  
 0.5% 15.9Ki 0.0% 945 [\_\_DATA]  
 0.4% 15.0Ki 0.4% 15.0Ki \_\_TEXT,\_\_cstring  
 0.0% 4 0.3% 11.3Ki [\_\_LINKEDIT]  
 0.0% 0 0.2% 8.12Ki \_\_DATA,\_\_bss  
 0.2% 8.01Ki 0.2% 8.01Ki [\_\_DATA\_CONST]  
 0.2% 7.43Ki 0.2% 7.43Ki Function Start Addresses  
 0.0% 0 0.2% 6.88Ki \_\_DATA,\_\_common  
 0.2% 6.08Ki 0.2% 6.08Ki Indirect Symbol Table  
 0.1% 4.59Ki 0.1% 4.59Ki \_\_DATA,\_\_la\_symbol\_ptr  
 0.1% 3.44Ki 0.1% 3.44Ki \_\_TEXT,\_\_stubs  
100.0% 3.44Mi 100.0% 3.45Mi TOTAL

### Оптимизации компилятора для уменьшения размера кода

Здесь приведены наиболее распространенные оптимизации компилятора, которые могут значительно уменьшить размер двоичного файла. Все упомянутые здесь флаги широко используются в индустрии.

* -Os: рассматривался ранее.
* -Wl,--strip-all (или не передавать флаг -g): этот флаг указывает компоновщику удалить раздел отладки.
* -fno-unroll-loops: отключает развертывание цикла, которое является одной из популярных оптимизаций производительности компилятора, увеличивающей размер кода.
* -fno-exceptions: удаляет код обработки исключений из двоичного файла. Обратите внимание, что это не всегда возможно, если есть код, который их «бросает».
* -lto (-flto): включение оптимизации времени компоновки с параметром -flto приводит к агрессивной оптимизации компилятора. Оптимизируются многие функции и глобальные переменные, девиртуализируются многие вызовы. Полученный двоичный файл быстрее и меньше одновременно. Могут быть значительные накладные расходы во время компиляции.

### Оптимизация исходного кода

#### Рефакторинг кода

Перемещение определений функций в файл .c/.cpp. Когда определения функций помещаются в заголовочные файлы, они дублируются в каждой единице трансляции, включающей заголовочный файл. Даже если в итоге остается только одно определение (благодаря One Definition Rule, ODR), эти функции могли быть вставлены в вызывающие их программы, и этот дополнительный объем кода сохранится в двоичном файле. Поэтому хорошей идеей является размещение определений функций в файлах .c/.cpp.

Помимо функций, которые были написаны разработчиками, существуют генерируемые компилятором функции, такие как конструкторы, деструкторы, перегрузки операторов и т. д. Даже эти функции могут вносить вклад в размер кода в зависимости от структуры типа и правил языка. Поэтому программисты могут явно определять эти методы в файле .cpp. Можно сделать либо определение «по умолчанию», либо явное. Например:

В файле test.h определен класс A:

class A {  
 a();  
 A(A const&);  
 ~A();  
};

В файле test.cpp определения инстанцированы:

A::A() = default;  
A(A const&) = default;  
A::~A() = default;

Подобно тому, как определения функций в заголовочных файлах увеличивают размер кода, шаблонные функции делают то же самое. Однако уменьшить их накладные расходы — нетривиальная задача. Часто бывает так, что некоторые типы используются чаще, чем другие. Для часто используемых типов мы можем явно инстанцировать их в файле .cpp. Например:

В файле **test.h** определен шаблон:

template<class T>  
struct a {  
void f(T t) { /\* \*/ }  
};

В файле test.cpp, шаблон явно инстанцирован:

template struct A<int>;

Явные инстанцирования также экономят время компиляции, поскольку инстанцирование происходит один раз. Для получения дополнительных идей по оптимизации исходного кода вы можете посмотреть презентацию Адитьи Кумара на международном форуме RISC‑V 2020: [«Code Size Compiler Optimizations and Techniques for Embedded Systems»](https://www.youtube.com/watch?v=6IuDWfuMEno).

#### Атрибуты функций

Атрибуты функций, которые уменьшают потенциал инлайнинга, могут помочь уменьшить размер кода. Например:

* \_\_attribute\_\_((cold))
* \_\_attribute\_\_((noinline))

Обратите внимание, что в некоторых случаях инлайнинг может уменьшить размер кода. В частности, при использовании крошечных функций инлайнинг устраняет накладные расходы на вызов функции, которые могут быть больше, чем размер тела самой функции. Рекомендуется использовать эти атрибуты в ограниченных случаях, так как они влияют на читабельность программ.

#### Уменьшение размера двоичного файла путем вынесения вычислений из двоичного файла

При хорошем знании оптимизации компилятора и требований языка программирования можно перенести вычисления из двоичного файла. Некоторые из выражений могут быть вычислены во время компиляции, в то время как некоторые другие могут быть отложены до выполнения. Оба подхода помогают уменьшить размер двоичного файла. Ниже приведены мотивирующие примеры.

Раннее вычисление: используя такие возможности языка, как constexpr, static\_assert из C++, некоторые выражения могут быть вычислены раньше, например:

constexpr auto gcd(int a, int b) {  
 while (b != 0){  
 auto t = b;  
 b = a % b;  
 a = t;  
 }  
 return a;  
}  
  
int main() {  
 int a = 11;  
 int b = 121;  
 int j = gcd(a, b);  
 constexpr int i = gcd(10, 12); // saves ‘2’ in the final assembly.  
 return i + j;  
}

Компилируя программу, представленную выше, используя команду g++ std=c++17 -fno-exceptions -S:

main:  
 mov edx, 121  
 mov eax, 11  
.L2: # inlined call to gcd(a, b)  
 mov ecx, edx  
 cdq  
 idiv ecx  
 mov eax, ecx  
 test edx, edx  
 jne .L2  
 add eax, 2 # Precomputed value of gcd(10,12)  
 ret

В ассемблере видно, что второй gcd был вычислен во время компиляции, но первый вызов gcd содержит весь код. Это происходит потому, что второй вызов функции gcd является constexpr. Подробнее о выражениях constexpr вы можете узнать на [веб-странице constexpr specifier](https://en.cppreference.com/w/cpp/language/constexpr).

#### Простые приёмы поиска мёртвого кода в бинарном файле

В любом крупном проекте, скорее всего, по разным причинам присутствует «мертвый код». Часть мертвого кода может быть удалена с помощью простых трюков. Например:

* Поиск тестирующего и отладочного кода, поставляемого в продакшн. Нетривиально найти код для тестирования/отладки, просматривая исходный код. Однако поиск в двоичном коде обеспечивает высокое соотношение сигнал/шум. nm можно использовать для поиска имен символов в двоичном коде.

nm <Binary> | grep -i "test\|debug"

* Поиск строк в бинарном файле с помощью инструмента strings. Как объяснялось ранее, strings выводит все C-строки, жестко закодированные в двоичном файле. Просмотрев строки, мы можем выяснить, почему та или иная строка оказалась в конечном бинарном файле.

## Характеристики производительности встраиваемых приложений

Определение и сфера применения встраиваемых систем со временем изменились. В то время как встраиваемые системы используются для обозначения вычислительных систем, выполняющих очень специфические задачи, во многих ситуациях это уже не так. Хотя большинство встраиваемых систем предназначены для выполнения ограниченного набора задач, в зависимости от области применения, сами задачи могут быть простыми или достаточно сложными. Встраиваемые системы могут иметь простые микроконтроллеры, или сложные цифровые сигнальные процессоры (DSP), или даже микропроцессоры. Даже при существенных различиях между системами, некоторые моменты в целом верны для всех:

* это маломощные устройства или устройства с батарейным питанием;
* они имеют ограниченный объем памяти;
* они нечасто обновляют приложения.

Могут быть и другие критерии, но эти три являются хорошей отправной точкой для понимания того, как мы можем оптимизировать эти приложения с помощью методов компилятора.

## Оптимизация энергопотребления

Существуют две точки зрения, как уменьшать энергопотребление во время работы приложения:

* выполнение/разработка инструкций с низким энергопотреблением;
* как можно более быстрое выполнение инструкций и переход в режим ожидания.

Каждый из подходов имеет свои достоинства и недостатки. Далее кратко описан каждый из них.

### Выполнение/разработка инструкций с низким энергопотреблением

Процессор имеет множество инструкций, с помощью которых можно выполнить одно и то же вычисление. Каждый тип инструкций потребляет разное количество системных ресурсов и подходит для определенных случаев. Например, операция с плавающей запятой может быть более дорогостоящей, чем целочисленные операции. В некоторых встраиваемых аппаратных средствах модули с плавающей запятой изначально отсутствуют, и для выполнения таких операций в редких случаях используются программные процедуры.

Компиляторы, насколько нам известно, не дают напрямую выбрать только инструкции с низким энергопотреблением. В результате этот подход применим в основном для инженеров по аппаратному обеспечению. В ограниченных ситуациях инженеры компиляторов могут воспользоваться преимуществами этого подхода, когда они имеют лучшее представление о процессоре и работающих приложениях; например, векторизация может быть отключена, поскольку векторные блоки часто потребляют больше энергии, чем скалярные инструкции.

Существуют также ситуации, когда система может иметь конфигурацию [big.LITTLE](https://www.arm.com/technologies/big-little). Более мощные процессоры используются только при увеличении потребности в вычислениях, в остальное время вычисления выполняются на маломощных процессорах. Если мы можем скомпилировать тяжелые для вычислений части кода для мощных процессоров (флаг -mtune), а остальные — для маломощных, мы можем воспользоваться этой возможностью. Эта стратегия потребует регулярного обновления флагов сборки в сочетании с оптимизацией под профиль (PGO).

### Как можно более быстрое выполнение инструкций и переход в режим ожидания

Некоторые процессоры могут иметь расширенные возможности динамического масштабирования напряжения и частоты. Таким образом, при отсутствии работы процессоры переходят в режим ожидания с низким энергопотреблением. В результате имеет смысл запускать приложения как можно быстрее и переложить ответственность за управление питанием на процессор. Чтобы запустить приложение как можно быстрее, мы можем использовать более высокие уровни оптимизации, такие как -O3, -Ofast; мы можем сочетать это с оптимизацией времени компоновки и оптимизацией с учетом профиля для достижения еще большей производительности.

## Оптимизация размера двоичного файла

Мы уже обсуждали уменьшение двоичного размера приложения в предыдущем разделе «[Оптимизации компилятора для уменьшения размера кода](#subsubsection-compiler-size-reduction)».

### Оптимизация приложений, которые не часто меняются

Если приложение меняется нечасто, то становится дешевле собирать профили времени выполнения и выполнять оптимизацию с учетом профиля. Приложение можно инструментировать с помощью флагов PGO, как обсуждалось ранее, и запустить его для обычных сценариев использования. Наличие комплексного набора тестов также может помочь в получении расширенных профилей производительности, что поможет компилятору эффективно оптимизировать приложение.

## Справочные материалы

* [Адитья Кумар — Оптимизация размера кода компилятора и методы для встраиваемых систем](https://www.youtube.com/watch?v=6IuDWfuMEno)
* [Адитья Кумар и Себастьян Поп — Анализ производительности и оптимизация стандартных библиотек C++](https://www.youtube.com/watch?v=OTCp_AkAyRQ)
* [Сообщество разработчиков Visual Studio — pragma optimize off работает не так, как ожидалось](https://developercommunity.visualstudio.com/t/192628900-pragma-optimize-off-is-not-working-as-ex/1091452)
* [nm(1) — страница руководства Linux](https://linux.die.net/man/1/nm)
* [Инструментирующие функции GCC](https://hacktalks.blogspot.com/2013/08/gcc-instrument-functions.html)
* [Нитин Кумар — Оптимизация с управлением по профилю (PGO) с использованием GCC на IBM AIX](https://developercommunity.visualstudio.com/t/192628900-pragma-optimize-off-is-not-working-as-ex/1091452)
* [Адитья Кумар — Анализ и оптимизация производительности](https://developercommunity.visualstudio.com/t/192628900-pragma-optimize-off-is-not-working-as-ex/1091452)
* [Винодха Рамасами, Пол Юань, Дехао Чен, Роберт Хундт — Feedback-Directed Optimizations in GCC with Estimated Edge Profiles from Hardware Event Sampling](https://research.google/pubs/pub36576/)

# Создание приложений RISC‑V

В этой главе мы узнаем, как использовать популярные системы компиляции (как LLVM, так и GCC) для создания приложений RISC‑V, а также как запускать приложения на симуляторах и эмуляторах. В этой главе представлены вводные материалы, которые помогут вам получить практический опыт создания и запуска приложений RISC‑V. Поскольку инструменты и их зависимости часто меняются, некоторые аспекты этого курса могут работать некорректно до тех пор, пока не будет произведено обновление курса. Помня об этом, мы приводим ссылки на официальные списки рассылки, репозитории GitHub и веб-сайты. Они могут оказаться весьма полезными, если вы захотите углубиться в какие-либо области разработки RISC‑V.

В этой главе мы обсудим:

* введение в тулчейн RISC‑V;
* где скачать тулчейны и симулятор;
* создание приложений с помощью этих тулчейнов;
* запуск приложений на симуляторе;
* запуск приложений на эмуляторе;
* полезные справочные материалы.

## Введение в системы компиляции RISC‑V

Есть две наиболее популярных системы компиляции RISC‑V.

1. GNU RISC‑V
2. LLVM RISC‑V

Обе системы предоставляют оптимизирующий компилятор, ассемблер, компоновщик и другие различные инструменты для создания приложений, работающих на машинах RISC‑V. Официальный [репозиторий RISC‑V на GitHub](https://github.com/riscv-collab/riscv-gcc) предоставляет исходный код для тулчейна riscv-gnu, тулчейн в собранном виде можно скачать по [ссылке](https://github.com/riscv-collab/riscv-gnu-toolchain) Компилятор RISC‑V обычно используется в качестве кросс-компилятора, поскольку многие процессоры RISC‑V используются для встраиваемых приложений с низким энергопотреблением. В настоящее время кросс-компиляторы RISC‑V можно запускать только на [целевых платформах ELF](https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats) таких, как Linux-машины.

Тулчейн RISC‑V LLVM недоступен у сторонних поставщиков, но его можно собрать из исходников, загрузив проект [llvm-project](https://github.com/llvm/llvm-project) с открытым кодом. Позже будет описан простой способ использования тулчейна llvm с gnu sysroot.

Симулятор RISC‑V можно загрузить из официального репозитория RISC‑V на GitHub: [riscv/riscv-isa-sim:Spike, симулятор RISC‑V ISA](https://github.com/riscv-software-src/riscv-isa-sim). Хотя платы RISC‑V легко доступны у нескольких производителей, разработку RISC‑V проще начать, используя симуляторы.

## Создание приложений с помощью тулчейна

Для создания приложения для RISC‑V использование инструментальной цепочки кросс-компилятора не отличается от любой другой системы разработки на основе кросс-компилятора. Требуется две вещи:

* система компиляции
* sysroot

В [Системы компиляции](#section-chapter1) мы описали систему компиляции и sysroot. Прежде чем научиться создавать приложения с помощью тулчейна RISC‑V, мы хотели бы представить интересное соглашение об именовании, которое используется для кросс-компиляторов.

### Соглашение об именовании компиляторов

Загрузив тулчейн GCC, в каталоге bin вы найдете двоичный файл riscv64-unknown-elf-gcc. Это тот же самый компилятор GCC со встроенной информацией о sysroot и платформе. Это соглашение — называть кросс-компиляторы таким образом. Обычно для именования используется формат arch-vendor-os-abi. Так, riscv64-unknown-elf-gcc означает, что это кросс-компилятор для RISC‑V 64 bit, и он будет генерировать двоичный файл elf, который может работать, например, на машинах Linux. Отличный справочник по соглашениям об именовании можно найти [здесь](http://web.eecs.umich.edu/~prabal/teaching/eecs373-f12/notes/notes-toolchain.pdf). В случаях, когда имя компилятора не содержит целевого триплета, для его получения можно использовать флаг -dumpmachine:

gcc -dumpmachine  
# x86\_64-linux-gnu

Имея под рукой компилятор riscv64-unknown-elf-gcc, файл можно скомпилировать следующими способами:

riscv64-unknown-elf-gcc -O2 -o a.out hello.c

riscv64-unknown-elf-gcc -O2 hello.c -mabi=lp64d -march=rv64ifd

Флаг -march используется для указания целевой субархитектуры, для которой будет сгенерирована сборка. Флаг -mabi используется для указания моделей данных[[4]](#footnote-111). Более подробную информацию о моделях данных можно найти в [этом разделе](https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models) Википедии о 64-битовых вычислениях.

С помощью тулчейна llvm двоичный файл может быть собран аналогичным образом. Предполагается, что sysroot находится в каталоге riscv64-unknown-elf:

clang test.c -c --sysroot riscv64-unknown-elf -target  
# riscv64-unknown-elf -march=rv64ifd

Приложения RISC‑V можно запускать как на устройствах, так и на симуляторах или эмуляторах. Мы кратко обсудим, как запустить приложение на симуляторе, а также на эмуляторе.

## Запуск приложений на симуляторе Spike

Одним из самых удобных способов запуска небольших приложений является использование симулятора RISC‑V. Шаги по сборке и установке достаточно просты. Для этого вам понадобятся следующие зависимости:

1. Linux-машина — создание образа Linux на других машинах нетривиально, поэтому для начала рекомендуется использовать Linux-машину.
2. Тулчейн RISC‑V: <https://github.com/riscv-software-src>
3. Spike, симулятор RISC‑V: <https://github.com/riscv-software-src/riscv-isa-sim>
4. pk, прокси-ядро RISC‑V: <https://github.com/riscv-software-src/riscv-pk>

Инструкции по запуску простого приложения hello-world на симуляторе Spike приведены в [README](https://github.com/riscv-software-src/riscv-isa-sim#compiling-and-running-a-simple-c-program) репозитория симулятора. Чтобы установить прокси-ядро, следуйте инструкциям в [README](https://github.com/riscv-software-src/riscv-pk#build-steps) репозитория pk. Для удобства вы можете установить и spike, и pk в тот же каталог, что и каталог тулчейна riscv64, указав путь к нему в качестве префикса установки для обоих.

## Запуск приложений на эмуляторе

Запуск приложения RISC‑V на эмуляторе обеспечивает большую гибкость, чем на симуляторе, но этапы установки требуют больше усилий. Чтобы запустить приложение RISC‑V на эмуляторе, необходимо иметь следующие зависимости:

1. Linux-машина — создание образа Linux на других машинах нетривиально, поэтому для начала рекомендуется использовать Linux-машину.
2. тулчейн RISC‑V: <https://github.com/riscv-software-src>
3. QEMU: git clone --depth 1 --branch v5.0.0 https://github.com/qemu/qemu
4. Linux: git clone --depth 1 --branch v5.4 https://github.com/torvalds/linux
5. Busybox: git clone --depth 1 git://git.busybox.net/busybox

Перечисленные выше ветки репозиториев являются предлагаемыми версиями, которые могут часто меняться. Вы можете выбрать и другие. Приведенная выше документация может устареть, если в какой-либо из зависимостей произойдут изменения. Проверьте страницу документации [Запуск 64- и 32-битных RISC‑V Linux на QEMU](https://risc-v-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html), чтобы узнать последние поддерживаемые версии. Если эта документация не работает, обязательно спросите в списке рассылки [linux-riscv](http://lists.infradead.org/pipermail/linux-riscv/).

### Сборка QEMU для RISC‑V

cd qemu  
./configure --target-list=riscv64-softmmu --prefix=/path/to/keep/qemu  
make -j $(nproc)  
make install

### Сборка Linux для целевой системы RISC‑V

cd linux  
make ARCH=riscv CROSS\_COMPILE=riscv64-unknown-linux-gnu- defconfig  
make ARCH=riscv CROSS\_COMPILE=riscv64-unknown-linux-gnu- -j $(nproc)

Убедитесь, что префикс кросс-компилятора совпадает с префиксом вашего тулчейна. В приведенном выше примере компилятор GCC — riscv64-unknown-linux-gnu-gcc, поэтому флаг CROSS\_COMPILE --riscv64-unknown-linux-gnu-.

### Сборка busybox

cd busybox  
CROSS\_COMPILE=riscv64-unknown-linux-gnu- make defconfig  
CROSS\_COMPILE=riscv64-unknown-linux-gnu- make -j $(nproc)

### Запуск образа Linux в QEMU

sudo /path/to/keep/qemu/bin/qemu-system-riscv64 -nographic -machine  
virt \  
 -kernel /path/to/linux/image -append "root=/dev/vda ro  
console=ttyS0" \  
 -drive file=busybox,format=raw,id=hd0 \  
 -device virtio-blk-device,drive=hd0

### Запуск приложения на QEMU на «голом железе»

/path/to/keep/qemu/bin/qemu-system-riscv64 -nographic -machine virt  
-kernel /path/to/binary -bios none

Дополнительные конфигурации QEMU для RISC‑V можно найти в [официальной документации](https://wiki.qemu.org/Documentation/Platforms/RISCV). Помимо симуляторов и эмуляторов, приложения RISC‑V можно запускать на виртуальных машинах, а также на имеющихся в продаже платах для разработки. Дополнительную документацию по отладке проблем с «голым железом» можно найти [здесь](https://embeddedinn.xyz/articles/tutorial/Adding-a-custom-peripheral-to-QEMU/). Вы можете установить виртуальную машину RISC‑V, как описано в документации [здесь](https://wiki.debian.org/RISC‑V).

## Справочные материалы

* [Tech: Toolchain & Runtime Subcommittee mailing list](mailto:tech-toolchain-runtime@lists.riscv.org)
* [кросс-компилятор GCC](https://wiki.osdev.org/GCC_Cross-Compiler)
* [64-битовые модели данных](https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models)
* [архив linux-riscv](https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models)
* [Running 64- and 32-bit RISC‑V Linux on QEMU](https://risc-v-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html)
* [Qemu: Документация/Платформы/RISCV](https://wiki.qemu.org/Documentation/Platforms/RISCV)
* [Debian — RISC‑V Wiki](https://wiki.debian.org/RISC‑V)

**Платы RISC‑V**

Страница [RISC‑V Exchange](https://riscv.org/exchange/) представляет собой коллекцию доступного физического оборудования в экосистеме RISC‑V. Этот список курируется сообществом.

**Ядра RISC‑V**

Страница [RISC‑V Exchange: Cores & SoCs](https://riscv.org/exchange/?_sft_exchange_category=core,cores) представляет собой коллекцию доступных IP-ядер систем на кристалле (СнК) в экосистеме RISC‑V.

**Поставщики тулчейнов и другого аппаратного и программного обеспечения:**

* [RISC‑V Software Collaboration](https://github.com/riscv-collab)
* [sifive/freedom-tools](https://github.com/sifive/freedom-tools/releases)
* [lowRISC](https://github.com/lowRISC)
* [stnolting/riscv-gcc-prebuilt](https://github.com/stnolting/riscv-gcc-prebuilt)
* [SiFive/Software](https://www.sifive.com/software)

# Методы отладки

Отладка — неотъемлемая часть жизненного цикла разработки программного обеспечения. Даже при самых надежных методах программирования возникают ошибки, и их отладка — не самое приятное занятие. В любом широко используемом программном обеспечении над кодом работают многие разработчики, и отладка занимает значительную часть их времени. Именно поэтому такие методологии разработки программного обеспечения, как [TDD](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (Test Driven Development, разработка через тестирование), [защитное программирование](https://en.wikipedia.org/wiki/Defensive_programming) и [контрактное программирование](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), приняты многими организациями. Некоторые языки программирования даже обеспечивают поддержку некоторых методологий; например, [D](https://ru.wikipedia.org/wiki/D_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) реализует функции [контрактного программирования](https://dlang.org/spec/contracts.html).

Отладка нативных приложений затруднена из-за семантики языка программирования, разнообразия зависимого программного обеспечения, зависимости от аппаратных характеристик, таких как модель потоков, модель памяти и т. д. Поэтому очень важно знать различные технологии отладки, доступные нам. В этой главе мы представим различные инструменты отладки и дадим краткое введение в каждый из них.

Сначала мы познакомимся с различными типами ошибок, которые встречаются в программном обеспечении, а затем изучим различные инструменты, облегчающие процесс отладки.

После знакомства с этой главой вы будете уметь:

* предотвращать появление многих ошибок с помощью защитных методов;
* использовать инструменты для своевременного обнаружения ошибок, позволяющие сэкономить много времени;
* использовать правильные инструменты и методы для отладки.

## Типы ошибок в программах

В целом, сбои в работе приложений можно разделить на три категории:

1. Ошибки программирования (синтаксические ошибки, семантические ошибки, ошибки времени выполнения).
2. Сбой системы (ошибки памяти, повреждение файла, сбой сети и т.д.).
3. Недействительные данные (некорректное имя файла, некорректные данные).

Некоторые ошибки трудно отнести к определенному классу, поскольку классификация может зависеть от точки отсчета. Например, повреждение данных в одном приложении может привести к тому, что зависимое приложение выдаст ошибку времени выполнения. В зависимости от типа сбоя применяются различные инструменты и методологии. В этой главе мы сосредоточимся на ошибках программирования, а в следующих разделах представим инструменты и методики, относящиеся к ним.

## Инструменты и методы отладки

При работе с тулчейном, настройке системы или разработке системного программного обеспечения часто используемые методы отладки можно разделить на четыре категории:

1. Отладка на основе инструментирования
2. Использование отладчика
3. Удалённая отладка с помощью gdb
4. Отладка на основе JTAG

### Отладка на основе инструментирования

Методы отладки на основе инструментирования — это методы вставки кода в программы вручную или программно. Это помогает получить представление о коде во время выполнения программы. Есть три наиболее широко используемых подхода:

1. Отладочная печать
2. Отладка на основе assertion’ов
3. Использование санитайзеров

Если отладочная печать и отладка на основе assertion’ов — одни из самых ранних методов отладки, то санитайзеры появились относительно недавно. Ниже мы кратко опишем каждую технику.

#### Отладочная печать

Простейшая техника отладки заключается в добавлении в программу набора операторов печати и наблюдении за выводимыми значениями. Вставка операторов печати в тщательно определенные точки программы может помочь отладить программу. Это один из самых ранних методов отладки, и он весьма полезен даже сегодня. Однако в ряде случаев этот подход не срабатывает, поскольку он плохо масштабируется. Если программист не обладает хорошим знанием кода, то при таком подходе трудно быстро найти проблемы.

#### Отладка на основе assertion

Чтобы установить пред- и постусловия в различных точках программы, обычно вставляются assertion’ы. Ошибка в assertion упрощает отладку. Это также помогает обнаруживать ошибки на ранней стадии разработки. Обычно assertion отключены в производственном коде, поэтому одним из первых шагов по отладке является включение assertion и запуск неудачного тестового сценария.

#### Санитайзеры

И clang, и GCC предоставляют инструментарий (санитайзеры), который может обнаружить несколько известных классов ошибок, таких как переполнение буфера, повреждение памяти и многих других. Существуют санитайзеры адресов, памяти, неопределенного поведения, потоков и прочие. Это обеспечивает автоматизированный способ обнаружения ошибок. Санитайзеры также требуют поддержки времени выполнения для поиска ошибок, а она доступна не на всех платформах. Например, RISC‑V в настоящее время поддерживает только санитайзер адресов и санитайзер потоков. [Этот документ](https://github.com/google/sanitizers) является источником достоверной информации для всей документации по санитайзерам.

На некоторых платформах [Valgrind](https://valgrind.org/) также может обнаружить ошибки, которые выявляются некоторыми санитайзерами, например, повреждение памяти или её утечки. К сожалению, в настоящее время Valgrind недоступен для платформ RISC‑V.

### Нативная отладка с помощью отладчика

Когда есть отладчик, становится очень удобно проверять программы, получать трассировку стека, значения объектов, состояние различных потоков и т.д. Обучение эффективному использованию отладчика, такого как gdb, значительно упрощает разработку программного обеспечения. Часто используемыми командами gdb являются:

* gdb --args <binary> <args> для запуска программы со списком аргументов.
* b FileName:linenum для установки точек останова
* bt для получения трассировки стека[[5]](#footnote-154)
* thread apply all bt для получения трассировки стека для всех потоков
* p variableName для печати значения переменной/объекта
* up для перехода вверх по кадру стека
* down для перехода вниз по кадру стека
* l для перечисления исходных строк кода вокруг текущего фрейма стека
* disassemble для отображения ассемблерного кода текущей функции
* n для выполнения следующей инструкции источника
* si для выполнения следующей машинной инструкции
* q для выхода из отладчика
* r для запуска программы

В сети существует множество источников, которые помогут вам начать работу с gdb, но вышеупомянутого списка команд достаточно для обычной повседневной отладки. Документацию по gdb можно найти в [здесь](https://www.gnu.org/software/gdb/documentation/).

### Удалённая отладка

gdb можно использовать для отладки программ, запущенных на другой машине. На удаленной машине запускается программа gdbserver, которая отвечает на запросы gdb клиентской машины. В некоторых случаях удаленная отладка может быть более удобной; например, она позволяет проверить программу на машине, где произошла ошибка, вместо того, чтобы пытаться повторить сценарий на компьютере разработчика. Это может сэкономить много времени. Удаленная отладка довольно популярна среди разработчиков мобильных приложений, когда приложение запускается на мобильном устройстве, а отладчик — на машине разработчика.

В случае заинтересованности вы можете начать работу со [статьи блога «Удаленная отладка с GDB»](https://developers.redhat.com/blog/2015/04/28/remote-debugging-with-gdb). Страница [руководства gdbserver(1) — Linux](https://man7.org/linux/man-pages/man1/gdbserver.1.html) также предлагает отличную документацию. В официальном [репозитории RISC‑V binutils](https://github.com/riscvarchive/riscv-binutils-gdb) есть исходный код gdbserver.

### Отладка на основе JTAG

На ранней стадии ввода в эксплуатацию аппаратного обеспечения отсутствуют программные возможности. Для отладки «голых» приложений или доступа к различным аппаратным блокам в качестве транспортного механизма используется JTAG (названный в честь Joint Test Action Group). Большинство современных аппаратных средств предоставляют возможность прямой трассировки инструкций и данных с помощью стандартного протокола JTAG, хотя у каждого производителя аппаратных средств может быть разный уровень поддержки. Он часто используется для отладки аппаратных неполадок. RISC‑V определяет стандартный интерфейс для внешней отладки, который включает в себя доступ к аппаратным потокам (hart) с самой первой инструкции, доступ к памяти, пошаговые инструкции и т. д.

## Справочные материалы

* [Корнельский университет, CS312 Лекция 26, Техники отладки](https://www.cs.cornell.edu/courses/cs312/2006fa/lectures/lec26.html)
* [Отладка с помощью GDB](https://www.sourceware.org/gdb/current/onlinedocs/gdb.html)
* [Тим Ньюсом — Внешняя отладка RISC‑V (она же JTAG-отладка)](https://riscv.org/wp-content/uploads/2016/01/Tues1030-RISC‑V-External-Debug.pdf)
* [Тим Ньюсом и Меган Вакс — Поддержка внешней отладки RISC‑V в версии 0.13.2](https://riscv.org/wp-content/uploads/2019/03/riscv-debug-release.pdf)
* [Five EmbedDev — Блог о встраиваемой RISC‑V](https://five-embeddev.com/riscv-debug-spec/latest/introduction.html#sec:intro)
* [Университет Сан-Франциско — Программирование «по контракту»](https://www.cs.usfca.edu/~parrt/course/601/lectures/programming.by.contract.html)
* [Отладка с помощью Gdb](https://github.com/riscv/riscv-isa-sim#debugging-with-gdb)
* [Константин Владимиров — Специальный выпуск: лекции по тулчейну, часть 4, линкеры и отладка](https://youtu.be/q2zztv9Z9i4)

# Список терминов

toolchain

система компиляции, тулчейн

high-level/low-level languages

языки высокого/низкого уровня

static analysis

статический анализ

instrumentation

инструментирование

introspection

интроспекция

compiler-driver

драйвер компилятора

bottleneck

узкое место

runtime profile

профиль исполнения

loop unrolling

развертывание циклов

inlining

встраивание функций или инлайнинг

instruction scheduling

планирование инструкций

register allocation

распределение регистров

benchmark

тест производительности

stack frame

кадр стека

buffer overflow

переполнение буфера

coverage file

файл покрытия

code layout

размещение кода

sampling profiler

семплирующий профилировщик

call site devirtualization

девиртуализация вызовов

instantiate

инстанцировать

dead code

мертвый код

comprehensive test suite

комплексный набор тестов

rich performance profiles

расширенные профили производительности

ELF targets

целевые платформы ELF

sub-architecture

субархитектура

SoC

система на кристалле, СнК

print debugging

отладочная печать

sanitizer

санитайзер

assertion

в курсе не переводится

breakpoint

точка останова или контрольная точка

backtrace (stack trace)

стектрейс, трассировка стека

single-stepping instructions

пошаговые инструкции

1. Примечание переводчика: перевод термина toolchain. Здесь и далее будет использоваться один из двух вариантов перевода: система компиляции или тулчейн. [↑](#footnote-ref-24)
2. Примечание переводчика: приведённые в курсе описание и иллюстрация канадской сборки могут лишь запутать читателя, рекомендуется ознакомиться с ней отдельно. [↑](#footnote-ref-38)
3. Подробнее об арифметике с плавающей запятой можно узнать в стандарте IEEE 754. [↑](#footnote-ref-64)
4. Примечание переводчика: не просто модель данных, а двоичный интерфейс приложения (ABI). [↑](#footnote-ref-111)
5. Примечание переводчика: исторически в gdb используется термин backtrace для обозначения трассировки стека (stack trace), поэтому команда и называется bt. [↑](#footnote-ref-154)