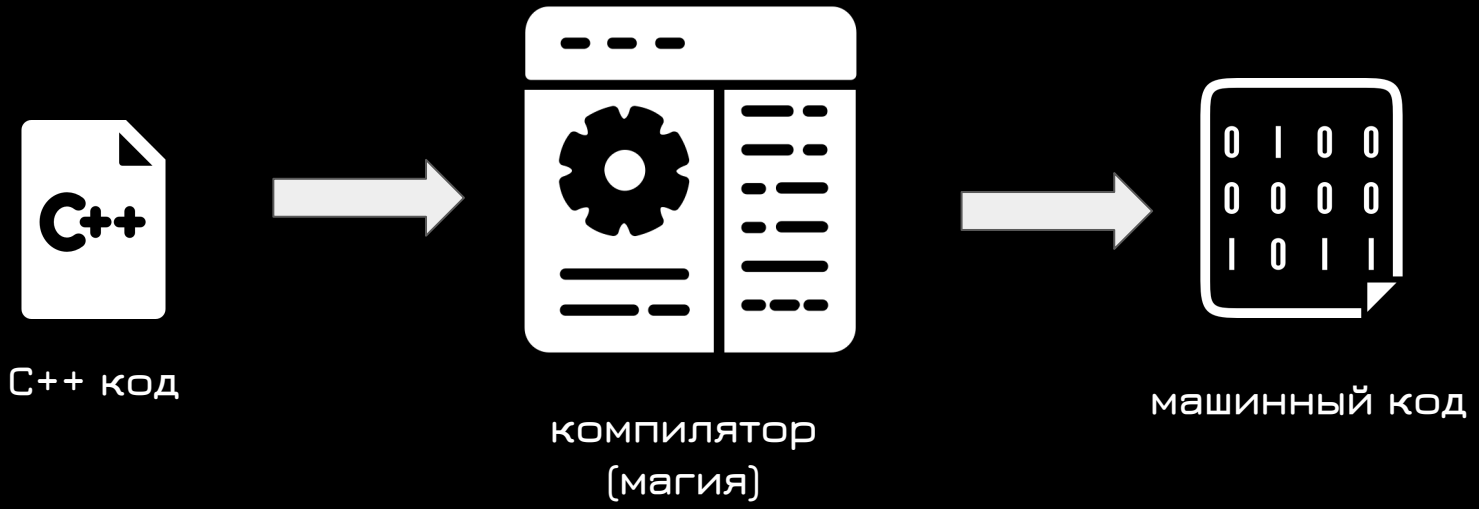
Как компилятор оптимизирует код и почему он умнее чем кажется?

Alex S

Оглавление

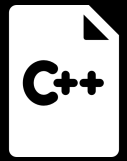
Как читаемый для человека текст становится понятен машине?

Компиляция кода

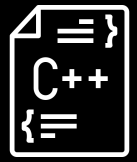


Какую магия делает компиляторы gcc и clang на x86_64?

Компиляция кода



C++ код

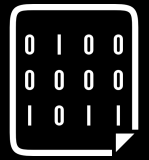


Препроцессинг

Компиляция

Ассемблирование

компоновка

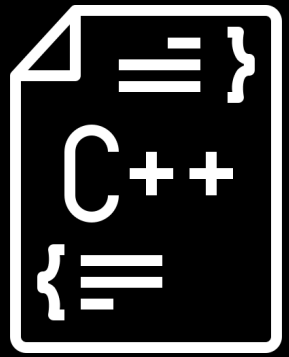


исполняемый файл

компилятор
(уже не магия)

Препроцессинг

- добавление include в код
- обработка макросов
- убирает комментарии
- обрабатывает #if #ifdef #ifndef

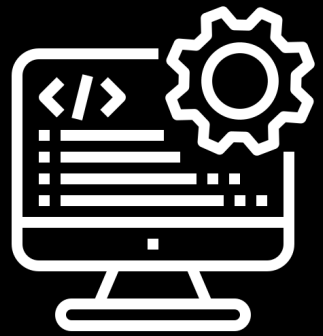



```
g++ -E main.cpp -o main.ii

clang++ -E main.cpp -o main.ii
```

Компиляция

преобразование кода в ассемблерный код



```
g++ -S main.cpp -o main.s
```

```
clang++ -S main.cpp -o main.s
```

Ассемблирование

Ассемблер преобразовывает ассемблерный код в машинный код, сохраняя его в *объектном файле*.


```
g++ -c main.s -o main.o

clang++ -c main.s -o main.o

objdump -d main.o > main.dump
```

Компоновка (линковка)

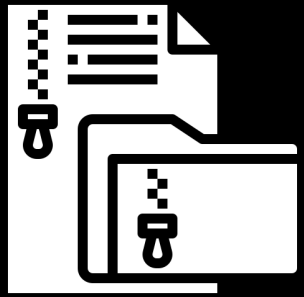
Компоновщик (линкер) позволяют объединять несколько файлов в исполняемые файлы или библиотеки. Объединение происходит в несколько шагов:

- Шаг 1: Разрешение символов

Программы определяют символы (переменные и функции) и ссылки на них. Определения символов хранятся в таблице символов.

- Шаг 2: Перемещение

Объединяет отдельные разделы кода и данных в единые разделы



```
g++ main.o -o main

clang++ main.o -o main
```

Что делает компилятор когда его просят оптимизировать код?

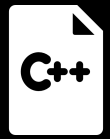
Флаги оптимизации -Ox

-O0 - без оптимизации
-O1 - компилятор пытается уменьшить размер кода и время выполнения, не выполняя никаких оптимизаций, которые занимают много времени компиляции.
-O2 - выполняет почти все поддерживаемые оптимизации, которые не требуют компромисса между космической скоростью и памятью.
-O3 - включает все оптимизации, указанные -O2, а также включает следующие флаги оптимизации: `-finline-functions, -fweb, -frename-registers, -funswitch-loops`

Как и когда компилятор оптимизирует код?

Подробнее о флагах оптимизации: https://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/Optimize-Options.html

Процесс компиляции



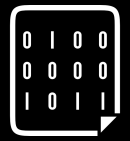
C++ код

Препроцессинг

Компиляция

Ассемблирование

компоновка



исполняемый файл

компилятор

Где тут оптимизации?

Процесс компиляции в gcc

Источник:
https://gcc.gnu.org/onlinedocs/gccint/Passes.html#Passes

Parsing pass

Parsing pass - парсит код используя любое промежуточное языковое представление, которое сочтет подходящим, например использует Abstract Syntax Trees.

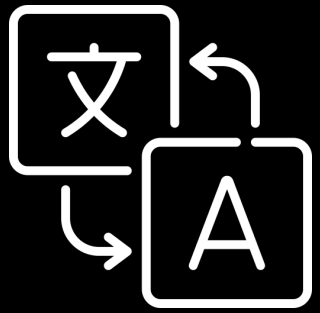
Результат парсинга

```
> g++ -fdump-tree-original -c main.cpp -o main
```

Источник:
https://gcc.gnu.org/onlinedocs/gccint/Parsing-pass.html#Parsing-pass

Gimplification pass

Gimplification pass - это причудливый термин для процесса преобразования функций в язык GIMPLE - представление, понятное не зависящим от языка частям компилятора.



Результат гимплификации дерева

```
> g++ -fdump-tree-gimple -c main.cpp -o main
```

Источник:
https://gcc.gnu.org/onlinedocs/gccint/Gimplification-pass.html#Gimplification-pass

Pass manager

Pass manager - его задача состоит в том, чтобы выполнять все отдельные проходы в правильном порядке

Источник:
https://gcc.gnu.org/onlinedocs/gccint/Pass-manager.html#Pass-manager

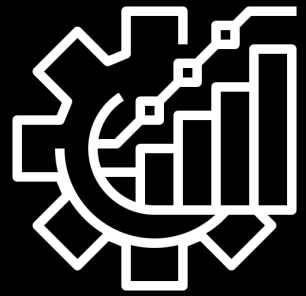
IPA pass

IPA pass - проходы межпроцедурнаой оптимизации, использующие информацию о графе вызовов для выполнения преобразований. IPA разделен на небольшие проходы:

- Small IPA passes:
  IPA remove symbols - анализ достижимости и удаления всех недостижимых узлов
- Regular IPA passes:
  IPA inline - опираясь на знания программы inline'ит функции
- Late IPA passes:
  IPA simd - создает соответствие SIMD для функций

Источник:
https://gcc.gnu.org/onlinedocs/gccint/IPA-passes.html#IPA-passes

Tree SSA passes

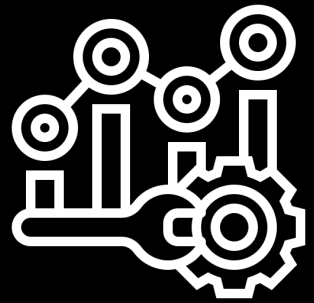


Tree SSA passes - этапы оптимизации дерева.

- Loop optimization:
  Loop splitting
  Vectorization
  Removal of empty loops
  Unrolling of small loops
- Return value optimization

Источник:
https://gcc.gnu.org/onlinedocs/gccint/Tree-SSA-passes.html#Tree-SSA-passes

RTL passes



RTL (Register Transfer Language) - язык инструкций, где каждая инструкция описываются, в значительной степени одна за другой, в алгебраической форме, которая описывает, что делает инструкция.
RTL passes - проходы, который выполняет оптимизацию RTL кода. Этот проход включает в себя:

- Loop optimization
- Jump bypassing
- Instruction combination

Сгенерировать RTL

```
> g++ -da -c main.cpp -o main.o
```

Источник:
https://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html#RTL-passes

Optimization info

Информация о примененных оптимизациях

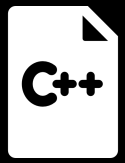
```
> gcc -fopt-info -Ox main.cpp -o main
```

Подробная информация о примененных оптимизациях на каждом проходе

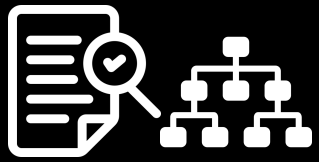
```
> g++ -fsave-optimization-record -O3  main.cpp -o main
```

Источник:
https://gcc.gnu.org/onlinedocs/gccint/Optimization-info.html#Optimization-info

Процесс компиляции в clang



C++ код



Parsing and Semantic Analysis

Code Generation

passes

low-level IR

ассемблерный код

компилятор

https://xiongyingfei.github.io/SA/2017/19_LLVM.pdf

Parsing and Semantic Analysis

Parsing and Semantic Analysis - синтаксический анализ входного файла, преобразующий токены препроцессора в дерево синтаксического анализа. Результатом этого этапа является “Абстрактное синтаксическое дерево” (AST).

Получение синтаксического дерева

```
> clang++ -cc1 -ast-dump main.cpp -o main.ast
```

Code Generation

Code Generation - На этом этапе AST преобразуется в низкоуровневый промежуточный код (известный как “LLVM IR”).

Получение читаемого llvm ir

```
> clang++ -S -emit-llvm main.cpp -o main.ll
```

Получение биткода

```
> clang++ -c -emit-llvm main.cpp -o main.bc
```

Оптимизация

Команда **opt** - это модульный оптимизатор и анализатор LLVM. Он принимает исходные файлы LLVM IR в качестве входных данных, выполняет указанные оптимизации или анализы на них, а затем выводит оптимизированный файл.

Оптимизация циклов

```
> opt --loops main.bc > main.opt.bc
```

Источник: https://llvm.org/docs/CommandGuide/opt.html

Получение ассемблера

Перевод из bitcode в ассемблер

```
> llc test.bc -o test.s
```

Источник:
https://subscription.packtpub.com/book/application-development/9781785285981/1/ch01lvl1sec14/converting-llvm-bitcode-to-target-machine-assembly

Optimization info

Информация о примененных оптимизациях

```
> clang++ -fsave-optimization-record=yaml -O3 main.cpp -o main
```

Источник: https://clang.llvm.org/docs/UsersManual.html

Проблема -Ox

Все оптимизации выполняются до этапа ассемблирования, а значит каждый полученный объектный файл оптимизировался независимо от остальных

Link Time Optimization

Link Time Optimization - оптимизация, выполняемая во время компоновки. Компоновщик знает обо всех единицах компиляции и может оптимизировать больше по сравнению с тем, что может сделать обычный компилятор.

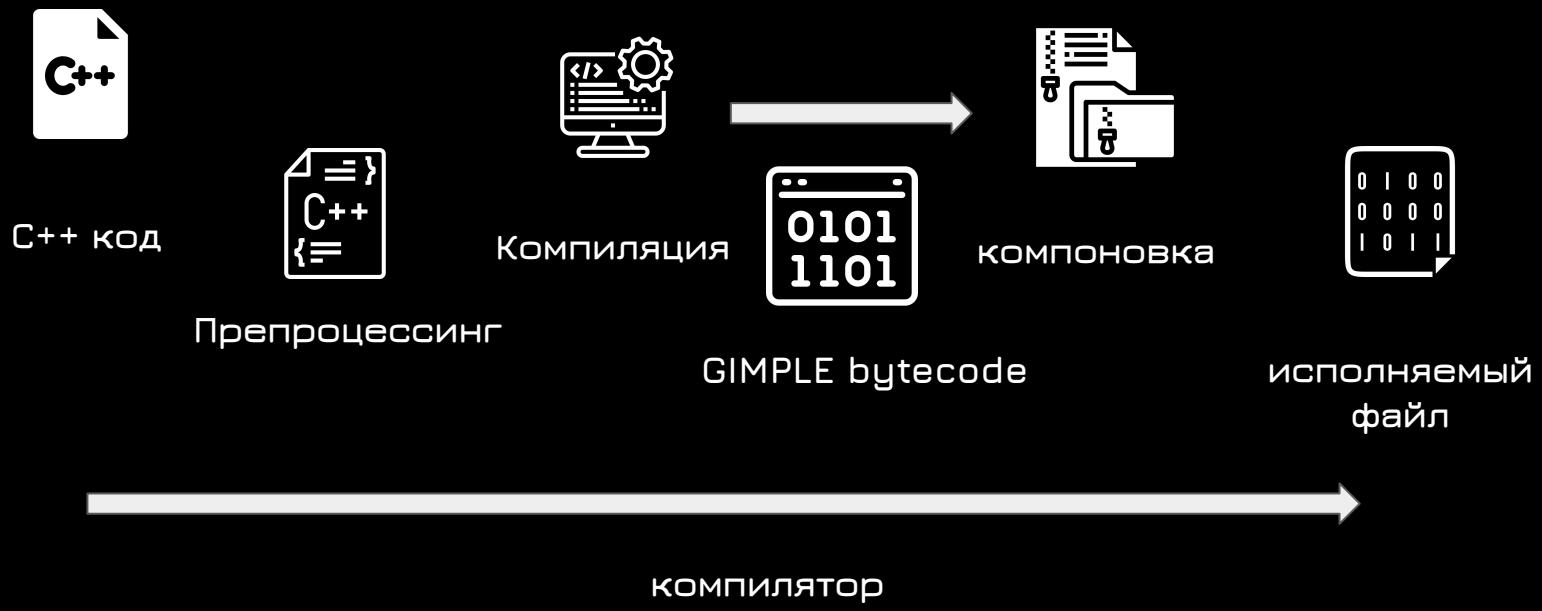
Сборка gcc + lto

```
> g++ -flto main.cpp foo.cpp -o main
```

Сборка clang + lto

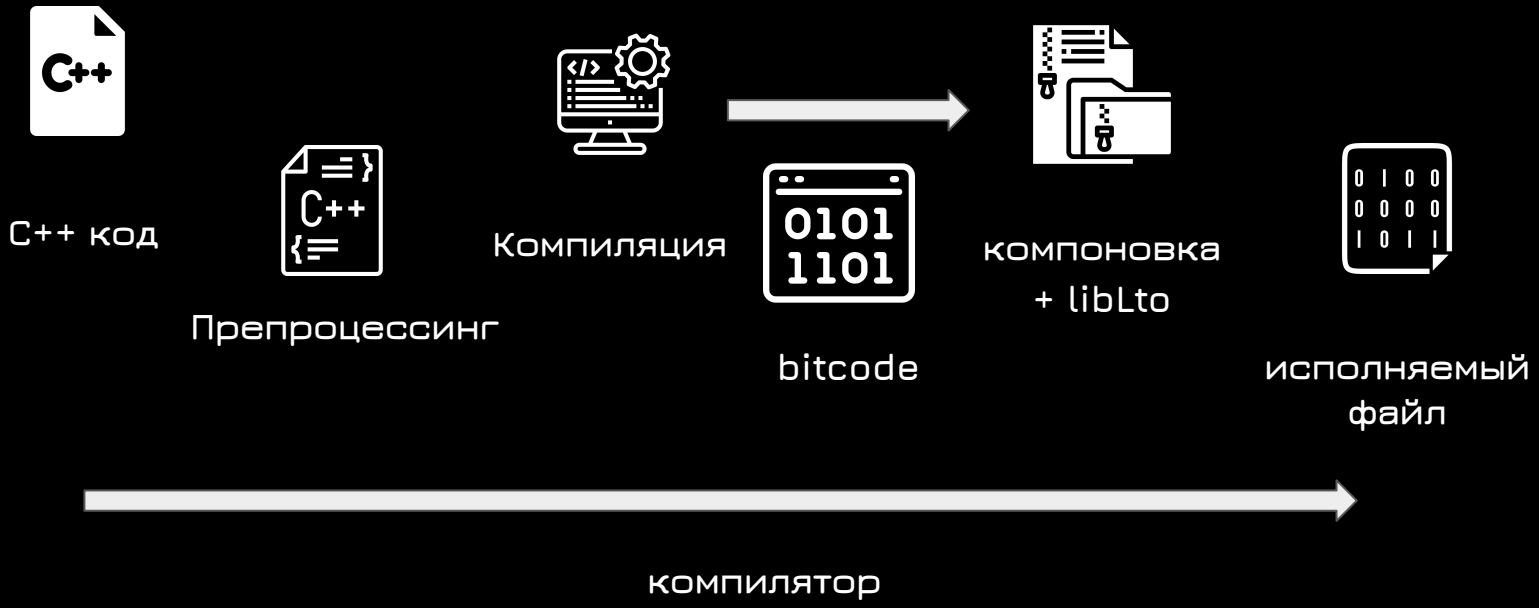
```
> clang -flto main.cpp foo.cpp -o main
```

Link Time Optimization в gcc



Источник:
https://gcc.gnu.org/onlinedocs/gccint/LTO.html#LTO

Link Time Optimization в clang



Проблема оптимизаций

Правильно ли компилятор подберет оптимизацию, не знаю о сценарии использования программы?

Решение: компилируем несколько раз!

Я уже говорил тебе ? Что такое - безумие, А ?
Безумие...- это точное повторение одного и того же действия раз за разом, в надежде на изменение - это и есть безумие.
(Far Cry 3)

Profile-guided optimization

Profile-guided optimization - техника оптимизации программы компилятором, которая использует результаты измерений тестовых запусков (профиль программы) для генерации оптимального кода.

Как генерируется профиль?

Profile-guided optimization

Инструментация - вставка дополнительных инструкций в код программы, обеспечивающих запись следующих параметров:

- Поток выполнения внутри функции
- Адреса косвенных вызов
- Аргументы функций работы с памятью

Виды инструментации:

- На уровне фронтенда (FE-level)
- На уровне промежуточного представления LLVM (IR-level)

Накладные расходы на инструментацию - зависит от структуры программы: чем больше ветвлений, а особенно операций с памятью и виртуальных вызовов, тем больше просадка по производительности

Profile-guided optimization в gcc

1. Сборка инструментированной версии

```
> gcc -fprofile-generate=<profile_dir> main.cpp -o main
```

2. Запуск инструментированной версии

```
> ./main
```

3. Сборка версии, оптимизированной с использованием профиля

```
> gcc -fprofile-use=<profile_dir> main.cpp
```

Profile-guided optimization в clang

1. Сборка инструментированной версии

```
> clang++ -O2 -fprofile-generate main.cpp -o main
```

2. Запуск инструментированной версии

```
> LLVM_PROFILE_FILE="code-%p.profraw" ./code
```

3. Объединение профилей и конвертирование их в формат, ожидаемый компилятором

```
> llvm-profdata merge -output=code.profdata code-*.profraw
```

4. Сборка версии, оптимизированной с использованием профиля

```
> clang++ -O2 -fprofile-use=code.profdata main.cc -o main
```