Содержание

gcc

GCC- GNU Compiler Collection

(https://www.opennet.ru/docs/RUS/gcc/)

Состав компилятора g++

cpp — препроцессор
as — ассемблер
g++ — компилятор
ld — линкер

Работа с GCC

(https://www.opennet.ru/docs/RUS/gcc/gcc1-2.html)

Общие опции

```
-Dname=value # Определить имя name в компилируемой программе, как значение value. Эффект такой же, как наличие строки #define name value в начале программы
-o filename # Использовать filename в качестве имени для создаваемого файла
-Wall # Вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы
```

Остановка на промежуточной стадии

```
-E # Остановиться после стадии препроцессирования
-S # Остановиться после собственно компиляции
-c # Компилировать или ассемблировать исходные файлы, но не линковать
```

Компиляция может включать до четырех стадий: препроцессирование, компиляцию, ассемблирование и линковку (всегда в этом порядке). Первые три стадии **применяются к отдельному** исходному файлу и заканчиваются получением объектного файла; линковка **объединяет все** объектные файлы.

Для любого имени входного файла суффикс определяет какая компиляция требуется:

```
file.c          # Исходный код на C, который нуждается в препроцессировании.
file.i          # Исходный код на C, который не нуждается в препроцессировании.
file.ii         # Исходный код на C++, который не нуждается в препроцессировании.
file.m          # Исходный код на Objective C. Заметим, что вам необходимо подключить библиотеку 'libobjc.a', чтобы заставить Objective C программу
```

```
работать.
file.h          # C заголовочный файл (не для компиляции или линковки).
file.cc         # \
file.cxx        #  \
file.cpp        # / Исходный код на C++, который нуждается в препроцессировании.
file.C          # /
file.s          # Ассемблерный код.
file.S          # Ассемблерный код, который нуждается в препроцессировании.
другие      # Объектный файл, который нужно отдать прямо на линковку. Так поступают с любым именем файла с нераспознанным суффиксом.
```

Так же язык можно указать явно: `-x язык`
Принимаются следующие языки: `c`; `objective-c` ; `c++` ; `c-header` ; `cpp-output` ; `c++-cpp-output` ; `assembler` ; `assembler-with-cpp`

Опции оптимизации

```
-O0 # Не оптимизировать
-O1 # Оптимизировать. Оптимизированная трансляции требует несколько больше времени и несколько больше памяти для больших функций
-O2 # Оптимизирует больше. GNU CC выполняет почти все поддерживаемые оптимизации, которые не включают уменьшение времени исполнения за счет увеличения длины кода
-O3 # Оптимизирует еще больше. Включает все оптимизации, определяемые -O2, а также включает опцию inline-functions.

# Если вы используете многочисленные -O опции с номерами или без номеров уровня, действительной является последняя такая опция.
```

Опции управления зависимостями

```
-lfoo # Использовать при компоновке динамическую библиотеку foo.so
# аналогично для статической библиотеки foo.a
-l:foo.so # Аналогично, но через двоеточие можно указать полное имя библиотеки
-Llibpath # Добавить к стандартным каталогам поиска библиотек путь libpath
-Iincludepath # Добавить к стандартным каталогам поиска заголовочных файлов путь includepath
```

Опция `-I/foo/include` не осуществляет сама инклудивание. Она лишь указывает папку, где нужно искать, поэтому нужен ещё и `#include`. То есть нужен и `-I/foo/include`, и `#include`, одного из них недостаточно.

Линковка. `-L/foo/lib` — это указание папки, где нужно искать бинарные файлы библиотеки, т. е. файлы .so и .a. `-lfoo` — это указание на то, что нужно, собственно, прилинковать эту библиотеку к результирующему бинарнику. Имя библиотеки, указанное в опции `-lfoo`, соединится с папкой, указанной в `-L/foo/lib` и получится `/foo/lib/libfoo` (в начало названия файла вставляется слово «lib»), затем сюда прибавится .so (или .a) и опционально номер версии и получится `/foo/lib/libfoo.so` или, скажем, `/foo/lib/libfoo.so.1`. Это и будет тем именем .so-файла, который будет искаться.

Так же, как и при компиляции (a.o из a.c), нужны обе опции `-L/foo/lib` и `-lfoo`. `-L/foo/lib` указывает, где искать. А `-lfoo` даёт окончательную команду на прилинковку.

Вместо `-lfoo` можно прямо написать целиком путь до файла библиотеки, который нужно слинковать, например, `/foo/lib/libfoo.so.1`. Тогда опция `-L/foo/lib` не нужна. Получится так: `cc -o a a.o /foo/lib/libfoo.so.1`

Примеры

Простая компиляция

Чтобы скомпилировать исходный код, необходимо компилятору gcc передать в качестве параметра исходный код (файл hello.cpp):

```
g++ hello.cpp -o prog
```

Дополнительный необязательный параметр `-o prog` указывает, что скомпилированный файл будет называться prog.exe. Если не указать этот параметр, то файл будет называться по умолчанию - a.exe.

Отдельная компиляция

C++ допускает раздельную компиляцию, а это означает, что каждый исходный файл может быть независимо скомпилирован в объектный файл. Эти объектные файлы затем можно соединить вместе, чтобы сформировать окончательный исполняемый файл. Это обеспечивает более быстрое время сборки при внесении изменений в один исходный файл, поскольку необходимо перекомпилировать только этот файл, а другие объектные файлы можно использовать повторно.

Пример раздельной компиляции и компоновки:

```
# Скомпилируем исходные файлы в объектные файлы
g++ -c main.cpp -o main.o
g++ -c funcs.cpp -o funcs.o

# Линкуем объектные файлы вместе, чтобы создать исполняемый файл
g++ main.o funcs.o -o my_program
```

Создание статической библиотеки

Допустим, у нас есть исходники с реализациями функций `first.cpp`, `second.cpp` и заголовочник `libmy.h` с объявлением ф-ий из исходников. Скомпилируем и заархивируем:

```
# компилируем
gcc -c first.cpp -o first.o
gcc -c second.cpp -o second.o
```

```
# архивируем:
ar crs libmy.a first.o second.o
# проверим:
file libmy.a # вывод: libmy.a: current ar archive
```

архиватор ar - утилита «склеивает» несколько файлов в один, не сжимая их содержимое.
`-c <filename>`: создать архив, если архив с именем «filename» не существует он будет создан, в противном случае файлы будут добавлены к имеющемуся архиву.
`-r`: Радает режим обновления архива, если в архиве файл с указанным именем уже существует, он будет удален, а новый файл дописан в конец архива.
`-s`: Добавляет (обновляет) индекс архива. В данном случае индекс архива это таблица, в которой для каждого определенного в архивируемых файлах символического имени (имени функции или блока данных) сопоставлено соответствующее ему имя объектного файла. Индекс архива необходим для ускорения работы с библиотекой - для того чтобы найти нужное определение, отпадает необходимость просматривать таблицы символов всех файлов архива, можно сразу перейти к файлу, содержащему искомое имя. Просмотреть индекс архива можно с помощью уже знакомой утилиты nm воспользовавшись её опцией `-s`.

Для создания индекса архива существует специальная утилита ranlib. Библиотеку libmy.a можно было сотворить и так:

```
ar cr libmy.a first.o second.o
ranlib libmy.a
```

Впрочем библиотека будет прекрасно работать и без индекса архива.

Подключение статической библиотеки

Допустим, у нас есть файл `main.cpp` , содержащий `#include «libmy.h»`. Соберём (компиляция+линковка) программу:

```
gcc -Wall -c main.c # компиляция программы
gcc -o mainprog main.o -L. -llibmy # линковка программы вместе с архивом с реализациями функций
```

Повторю:
`-L/путь/к/каталогу/с/библиотеками` указывает путь к каталогу содержащему подключаемые библиотеки.
`-l<name>` имя библиотеки задается как «name» без приставки «lib»

Компиляция либы:

```
// компиляция lib.cpp в объектный файл
g++ -c lib.cpp -o lib.o
// архивируем в статическую библиотеку
ar rcs libmylib.a lib.o
```

Использование либы:

```
// компилим файл main с подключением либы
g++ main.cpp -o prog -L. -lmylib
```

Здесь: - `-L .` указывает компилятору искать библиотеку в текущем каталоге. - `-lmylib` говорит компилятору использовать библиотеку `libmylib.a`.

Эти шаги создадут исполняемый файл `prog`, содержащий код из `main.cpp`, а также функции из вашей статической библиотеки `libmylib.a`.

Если вы хотите создать динамическую библиотеку, то процесс будет немного отличаться. Для этого вместо создания статической библиотеки (`libmylib.a`) вы создадите разделяемую библиотеку (`libmylib.so` или `.dll` в зависимости от вашей операционной системы и компилятора) и при компиляции программы вместо флага `-lmylib` вы будете использовать флаг `-lmylib.so` или `-lmylib.dll`, соответственно.

Создание динамической библиотеки

Набор исходных файлов аналогичен примеру сверху.

```
gcc -fPIC -c first.cpp
gcc -fPIC -c second.cpp
gcc -shared -o libhello.so.2.4.0.5 -Wl,-soname,libhello.so.2 first.o
second.o
```

`-fPIC` - требует от компилятора, при создании объектных файлов, порождать позиционно-независимый код (PIC - Position Independent Code), его основное отличие в способе представления адресов. Вместо указания фиксированных (статических) позиций, все адреса вычисляются исходя из смещений заданных в глобальной таблицы смещений (**g**lobal **o**ffset **t**able - GOT). Формат позиционно-независимого кода позволяет подключать исполняемые модули к коду основной программы в момент её загрузки. Соответственно, основное назначение позиционно-независимого кода - создание динамических (разделяемых) библиотек.
`-shared` - указывает `gcc`, что в результате должен быть собран не исполняемый файл, а «разделяемый объект» - динамическая библиотека.
`-Wl` - Обычно, компилятор сам вызывает линковщик и передает ему параметры по своему усмотрению. Этот параметр позволяет передать линковщику опции руками. Общий вид: `gcc -Wl,-option,value1,value2…` что означает передать линковщику (`-Wl`) опцию `-option` с аргументами `value1, value2`.

`-soname` - опция линковщика (передаваемая через `-W`) Fix Me!

(https://uzverss.livejournal.com/57883.html)

Подключение динамической библиотеки

Fix Me!