2. Специфика сборки программ в Linux

2.1. Hello World

Чтобы сразу начать программировать, создадим еще один клон известной программы "Hello World". Что делает эта программа, вы знаете. Откройте свой любимый текстовый редактор и наберите в нем следующий текст:

```
/* hello.c */
#include <stdio.h>

int main (void)
{
        printf ("Hello World\n");
}
```

Я назвал свой файл **hello.c**. Вы можете назвать как угодно, сохранив суффикс **.c**. Содержимое файла hello.c - это *исходный код* программы ('program source', 'source code' или просто 'source'). А hello.c - это *исходный файл* программы ('source file'). Hello World - очень маленькая программа, исходный код которой помещается в одном файле. В "настоящих" программах, как правило, исходный код разносится по нескольким файлам. В больших программах исходных файлов может быть больше сотни.

Наш исходный код написан на языке программирования C. Языки программирования были придуманы для того, чтобы программист мог объяснить компьютеру, что делать. Но вот беда, компьютер не понимает ни одного языка программирования. У компьютера есть свой язык, который называют *машинным кодом* или *исполняемым кодом* ('executable code'). Написать Hello World в машинном коде можно, но серьезные программы на нем не пишутся. Исполняемый код не только сложный по своей сути, но и очень неудобный для человека. Программа, которую можно написать за один день на языке программирования будет писаться целый год в машинном коде. Потом программист сойдет с ума. Чтобы этого не случилось, был придуман компилятор ('compiler'), который переводит исходный код программы в исполняемый код. Процесс перевода исходного кода программы в исполняемый код называют *компиляцией*.

Чтобы откомпилировать наш Hello World достаточно набрать в командной строке следующее заклинание:

```
$ gcc -o hello hello.c
$
```

Если исходный код написан без **синтаксических** ошибок, то компилятор завершит свою работу без каких-либо сообщений. Молчание - знак повиновения и согласия. Набрав команду **ls** вы тут же обнаружите новый файл с именем hello. Этот файл содержит исполняемый код программы. Такие файлы называют исполняемыми файлами ('executable files') или *бинарниками* ('binary files').

Вы наверняка догадались, что опция -o компилятора gcc указывает на то, каким должно быть имя **выходного** файла. Как вы позже узнаете, выходным файлом может быть не только бинарник. Если не указать опцию -o, то бинарнику, в нашем случае, будет присвоено имя a.out.

Осталось только запустить полученный бинарник. Для этого набираем в командной строке следующую команду:

```
$ ./hello
Hello World
$
```

Когда мы набираем в командной строке путь к бинарнику, мы, в реальности сообщаем оболочке, что надо выполнить программу. Оболочка "передает" бинарник ядру операционной системе, а ядро системы особым шаманским способом отдает программу на выполнение процессору. Затем, если программа не была запущена в фоновом режиме, то оболочка ждет от ядра сообщения о том, что программа выполнилась. Получив такое сообщение, оболочка выдает приглашение на ввод новой команды. Вы можете еще раз набрать **./hello** и процедура повторится. В нашем случае программа выполняется очень быстро, и новое приглашение командной строки "вылетает" практически сразу.

Мы рассмотрели идеальный случай, когда программа написана без **синтаксических** ошибок. Попробуем намеренно испортить программу таким образом, чтобы она не отвечала канонам языка С. Для этого достаточно убрать точку с запятой в конце вызова функции **printf()**:

```
printf ("Hello World\n")
```

Теперь, если попытаться откомпилировать программу, то компилятор выругается, указав нам на то, что он считает неправильным:

```
$ gcc -o hello hello.c
hello.c: In function 'main':
hello.c:7: error: syntax error before '}' token
$
```

В первой строке говорится, что в файле hello.c (у нас он единственный) в теле функции main() что-то произошло. Вторая строка сообщает, что именно произошло: седьмая строка файла hello.c вызвала ошибку (error). Далее идет расшифровка: синтаксическая ошибка перед закрывающейся фигурной скобкой.

Заглянув в файл hello.c мы с удивлением обнаружим, что нахулиганили мы не в седьмой, а в шестой строке. Дело в том, что компилятор обнаружил нелады только в седьмой строке, но написал 'before' (до), что означает "прокручивай назад".

Естественно, пока мы не исправим ошибку, ни о каком бинарнике не может идти и речи. Если мы удалим старый бинарник hello, доставшийся нам от прошлой компиляции, то увидим, что компиляция испорченного файла не даст никакого результата. Однако иногда компилятор может лишь "заподозрить" что-то неладное, потенциально опасное для нормального существования программы. Тогда вместо 'error' пишется 'warning' (предупреждение), и бинарник все-таки появляется на свет (если в другом месте нет явных ошибок). Не следует игнорировать предупреждения, за исключением тех случаев, когда вы на 100% знаете, что делаете.

Парадокс программирования заключается в том, что можно наделать кучу ошибок (уже не синтаксических, как в нашем случае, а смысловых) по всем правилам языка программирования. В таком случае компилятор выдает бинарник, который делает не то, что мы хотели. В таком случае программу приходится *отлаживать*. Отладка - это обычное дело

при написании любой достаточно сложной программы. Не ошибается только тот, кто ничего не делает.

2.2. Мультифайловое программирование

Как я уже говорил, если исходный код сколько-нибудь серьезной программы уместить в одном файле, то такой код станет просто нечитаемым. К тому же если программа компилируется достаточно долго (особенно это относится к языку C++), то после исправления одной ошибки, нужно перекомпилировать весь код.

Куда лучше разбросать исходный код по нескольким файлам (осмысленно, по какому-нибудь критерию), и компилировать каждый такой файл отдельно. Как вы вскоре узнаете, это очень даже просто.

Давайте сначала разберемся, как из исходного файла получается бинарник. Подобно тому как гусеница не сразу превращается в бабочку, так и исходный файл не сразу превращается в бинарник. После компиляции создается *объектный код*. Это исполняемый код с некоторыми "вкраплениями", из-за которых объектный код еще не способен к выполнению. Сразу в голову приходит стиральная машина: вы ее только что купили и она стоит у вас дома в коробке. В таком состоянии она стирать не будет, но вы все равно рады, потому что осталось только вытащить из коробки и подключить.

Вернемся к объектному коду. Эти самые "вкрапления" (самое главное среди них - таблица символов) позволяют объектному коду "пристыковываться" к другому объектному коду. Такой фокус делает *компоновщик* (*линковщик*) - программа, которая объединяет объектный код, полученный из "разных мест", удаляет все лишнее и создает полноценный бинарник. Этот процесс называется *компоновкой* или *линковкой*.

Итак, чтобы откомпилировать мультифайловую программу, надо сначала добыть объектный код из каждого исходного файла в отдельности. Каждый такой код будет представлять собой *объектный модуль*. Каждый объектный модуль записывается в отдельный *объектный файл*. Затем объектные модули надо скомпоновать в один бинарник.

В Linux в качестве линковщика используется программа **ld**, обладающая приличным арсеналом опций. К счастью gcc самостоятельно вызывает компоновщик с нужными опциями, избавляя нас от "ручной" линковки.

Попробуем теперь, вооружившись запасом знаний, написать мультифайловый Hello World. Создадим первый файл с именем main.c:

```
/* main.c */

int main (void)
{
        print_hello ();
}
```

Теперь создадим еще один файл hello.c со следующим содержимым:

```
/* hello.c */
#include <stdio.h>

void print_hello (void)
{
        printf ("Hello World\n");
}
```

Здесь функция main() вызывает функцию print_hello(), находящуюся в другом файле. Функция print_hello() выводит на экран заветное приветствие. Теперь нужно получить два объектных файла. Опция -c компилятора gcc заставляет его отказаться от линковки после компиляции. Если не указывать опцию -o, то в имени объектного файла расширение .c будет заменено на .o (обычные объектные файлы имеют расширение .o):

```
$ gcc -c main.c
$ gcc -c hello.c
$ ls
hello.c  hello.o  main.c  main.o
$
```

Итак, мы получили два объектных файла. Теперь их надо объединить в один бинарник:

```
$ gcc -o hello main.o hello.o
$ ls
hello*  hello.c  hello.o  main.c  main.o
$ ./hello
Hello World
$
```

Компилятор "увидел", что вместо исходных файлов (с расширением .c) ему подбросили объектные файлы (с расширением .o) и отреагировал согласно ситуации: вызвал линковщик с нужными опциями.

Давайте разберемся, что же все-таки произошло. В этом нам поможет утилита **nm**. Я уже оговорился, что объектные файлы содержат *таблицу символов*. Утилита nm как раз позволяет посмотреть эту таблицу в читаемом виде. Те, кто пробовал программировать на ассемблере знают, что в исполняемом файле буквально все (функции, переменные) стоит на своей позиции: стоит только вставить или убрать из программы один байт, как программа тут же превратиться в груду мусора из-за смещенных позиций (адресов). У объектных файлов особая роль: они хранят в таблице символов имена некоторых позиций (глобально объявленных функций, например). В процессе линковки происходит стыковка имен и пересчет позиций, что позволяет нескольким объектным файлам объединиться в один бинарник. Если вызвать nm для файла hello.o, то увидим следующую картину:

```
$ nm hello.o
         U printf
00000000 T print_hello
$
```

О смысловой нагрузке нулей и литер U,T мы будем говорить при изучении библиотек. Сейчас же важным является то, что в объектном файле сохранилась информация об использованных именах. Своя информация есть и в файле main.o:

```
$ nm main.o
00000000 T main
         U print_hello
$
```

Таблицы символов объектных файлов содержат общее имя print_hello. В процессе линковки

высчитываются и подставляются в нужные места адреса, соответствующие именам из таблицы. Вот и весь секрет.

2.3. Автоматическая сборка

В предыдущем разделе для создания бинарника из двух исходных файлов нам пришлось набрать три команды. Если бы программу пришлось отлаживать, то каждый раз надо было бы вводить одни и те же три команды. Казалось бы выход есть: написать сценарий оболочки. Но давайте подумаем, какие в этом случае могут быть недостатки. Во-первых, каждый раз сценарий будет компилировать **все** файлы проекта, даже если мы исправили только один из них. В нашем случае это не страшно. Но если речь идет о десятках файлов! Во-вторых, сценарий "намертво" привязан к конкретной оболочке. Программа тут же становится менее переносимой. И, наконец, простому скрипту не хватает функциональности (задание аргументов сборки и т. п.), а хороший скрипт (с многофункциональными прибамбасами) плохо модернизируется.

Выход из сложившейся ситуации есть. Это утилита **make**, которая работает со своими собственными сценариями. Сценарий записывается в файле с именем Makefile и помещается в репозиторий (рабочий каталог) проекта. Сценарии утилиты make просты и многофункциональны, а формат Makefile используется повсеместно (и не только на Unix-системах). Дошло до того, что стали создавать программы, генерирующие Makefile'ы. Самый яркий пример - набор утилит GNU Autotools. Самое главное преимущество make - это "интеллектуальный" способ рекомпиляции: в процессе отладки make компилирует только измененные файлы.

То, что выполняет утилита make, называется *сборкой* проекта, а сама утилита make относится к разряду *сборщиков*.

Любой Makefile состоит из трех элементов: *комментарии*, *макроопределения* и *целевые связки* (или просто *связки*). В свою очередь связки состоят тоже из трех элементов: *цель*, *зависимости* и *правила*.

Сценарии make используют однострочные комментарии, начинающиеся с литеры # (решетка). О том, что такое комментарии и зачем они нужны, объяснять не буду.

Макроопределения позволяют назначить имя практически любой строке, а затем подставлять это имя в любое место сценария, где должна использоваться данная строка. Макросы Makefile схожи с макроконстантами языка C.

Связки определяют: 1) что нужно сделать (цель); 2) что для этого нужно (зависимости); 3) как это сделать (правила). В качестве цели выступает имя или макроконстанта. Зависимости - это список файлов и целей, разделенных пробелом. Правила - это команды передаваемые оболочке.

Теперь рассмотрим пример. Попробуем составить сценарий сборки для рассмотренного в предыдущем разделе мультифайлового проекта Hello World. Создайте файл с именем Makefile:

```
# Makefile for Hello World project

hello: main.o hello.o
	gcc -o hello main.o hello.o

main.o: main.c
	gcc -c main.c

hello.o: hello.c
	gcc -c hello.c
```

```
clean:
	rm -f *.o hello
```

Обратите внимание, что в каждой строке перед вызовом gcc, а также в строке перед вызовом rm стоят **табуляции**. Как вы уже догадались, эти строки являются правилами. Формат Makefile требует, чтобы каждое правило начиналось с табуляции. Теперь рассмотрим все по порядку.

Makefile может начинаться как с заглавной так и со строчной буквы. Но рекомендуется все-таки начинать с заглавной, чтобы он не перемешивался с другими файлами проекта, а стоял "в списке первых".

Первая строка - комментарий. Здесь можно писать все, что угодно. Комментарий начинается с символа # (решетка) и заканчивается символом новой строки. Далее по порядку следуют четыре связки: 1) связка для компоновки объектных файлов main.o и hello.o; 2) связка для компиляции main.c; 3) связка для компиляции hello.c; 4) связка для очистки проекта.

Первая связка имеет цель hello. Цель отделяется от списка зависимостей двоеточием. Список зависимостей отделяется от правил символом новой строки. А каждое правило начинается на новой строке с символа табуляции. В нашем случае каждая связка содержит по одному правилу. В списке зависимостей перечисляются через пробел вещи, необходимые для выполнения правила. В первом случае, чтобы скомпоновать бинарник, нужно иметь два объектных файла, поэтому они оказываются в списке зависимостей. Изначально объектные файлы отсутствуют, поэтому требуется создать целевые связки для их получения. Итак, чтобы получить main.o, нужно откомпилировать main.c. Таким образом файл main.c появляется в списке зависимостей (он там единственный). Аналогичная ситуация с hello.o. Файлы main.c и hello.c изначально существуют (мы их сами создали), поэтому никаких связок для их создания не требуется.

Особую роль играет целевая связка clean с пустым списком зависимостей. Эта связка очищает проект от всех автоматически созданных файлов. В нашем случае удаляются файлы main.o, hello.o и hello. Очистка проекта бывает нужна в нескольких случаях: 1) для очистки готового проекта от всего лишнего; 2) для пересборки проекта (когда в проект добавляются новые файлы или когда изменяется сам Makefile; 3) в любых других случаях, когда требуется полная пересборка (напрмиер, для измерения времени полной сборки).

Теперь осталось запустить сценарий. Формат запуска утилиты make следующий:

```
make [опции] [цели...]
```

Опции make нам пока не нужны. Если вызвать make без указания целей, то будет выполнена первая попавшаяся связка (со всеми зависимостями) и сборка завершится. Нам это и требуется:

```
$ make
gcc -c main.c
gcc -c hello.c
gcc -o hello main.o hello.o
$ ls
hello*  hello.c  hello.o  main.c  main.o  Makefile
$ ./hello
Hello World
$
```

В процессе сборки утилита make пишет все выполняемые правила. Проект собран, все

работает.

Теперь давайте немного модернизируем наш проект. Добавим одну строку в файл hello.c:

```
/* hello.c */
#include <stdio.h>

void print_hello (void)
{
        printf ("Hello World\n");
        printf ("Goodbye World\n");
}
```

Теперь повторим сборку:

```
$ make
gcc -c hello.c
gcc -o hello main.o hello.o
$ ./hello
Hello World
Goodbye World
$
```

Утилита make "пронюхала", что был изменен только hello.c, то есть компилировать нужно только его. Файл main.o остался без изменений. Теперь давайте очистим проект, оставив одни исходники:

```
$ make clean
rm -f *.o hello
$ ls
hello.c  main.c  Makefile
$
```

В данном случае мы указали цель непосредственно в командной строке. Так как целевая связка clean содержит пустой список зависимостей, то выполняется только одно правило. Не забывайте "чистить" проект каждый раз, когда изменяется список исходных файлов или когда изменяется сам Makefile.

2.4. Модель КИС

Любая программа имеет свой *репозиторий* - рабочий каталог, в котором находятся исходники, сценарии сборки (Makefile) и прочие файлы, относящиеся к проекту. Репозиторий рассмотренного нами проекта мультифайлового Hello World изначально состоит из файлов main.c, hello.c и, собственно, Makefile. После сборки репозиторий дополняется файлами main.o, hello.o и hello. Практика показывает, что правильная организация исходного кода в репозитории не только упрощает модернизацию и отладку, но и предотвращает возможность появления многих ошибок.

Модель КИС (Клиент-Интерфейс-Сервер) - это элегантная концепция распределения исходного кода в репозитории, в рамках которой все исходники можно поделить на *клиенты*, *интерфейсы* и *серверы*.

Итак, сервер предоставляет услуги. В нашем случае это могут быть функции, структуры, перечисления, константы, глобальные переменные и проч. В языке C++ это чаще всего

классы или иерархии классов. Любой желающий (клиент) может воспользоваться предоставленными услугами, то есть вызвать функцию со своими фактическими параметрами, создать экземпляр структуры, воспользоваться константой и т. п. В C++, как правило, клиент использует класс как тип данных и использует его члены.

Часто бывает, что клиент сам становится сервером, точнее начинает играть роль промежуточного сервера. Хороший пример - наш мультифайловый Hello World. Здесь функция print_hello() (клиент) пользуется услугами стандартной библиотеки языка C (сервер), вызывая функцию printf(). Однако в дальнейшем функция print_hello() сама становится сервером, предоставляя свои услуги функции main(). В языке C++ довольно часто клиент создает производный класс, который наследует некоторые механизмы базового класса сервера. Таким образом клиент сам становится сервером, предоставляя услуги своего производного класса.

Клиент с сервером должны "понимать" друг друга, иначе взаимодействие невозможно. *Интерфейс* (протокол) - это условный набор правил, согласно которым взаимодействуют клиент и сервер. В нашем случае (мультифайловый Hello World) интерфейсом (протоколом) является общее имя в таблице символов двух объектных файлов. Такой способ взаимодействия может привести к неприятным последствиям. Клиент (функция main()) не знает ничего, кроме имени функции print_hello() и, наугад вызывает ее без аргументов и без присваивания. Иначе говоря, клиент не знает до конца правил игры. В нашем случае **прототип** функции print_hello() неизвестен.

Обычно для организации интерфейсов используются *объявления* (прототипы), которые помещаются чаще всего в *заголовочные файлы*. В языке C это файлы с расширением .h; в языке C++ это файлы с раширением .h, .hpp или без расширения. Некоторые "всезнайки" **ошибочно называют заголовочные файлы библиотеками** и умудряются учить этому других. Забегая вперед скажу, что библиотека - это просто **коллекция скомпонованных особым образом объектных файлов**, а заголовочный файл - это **интерфейс**. Основная разница между библиотеками и заголовочными файлами в том, что библиотека - это объектный (почти исполняемый) код, а заголовочный файл - это исходный код. Включая в программу заголовочный файл директивой #include мы соглашаемся работать с сервером (будь то библиотека или простой объектный файл) по его протоколу: если сервер сказал, что функция вызывается без аргументов, то она и будет вызываться без аргументов, иначе компилятор костьми ляжет, но не даст откомпилировать "незаконный вызов".

Вернемся к Hello World. В таком виде, как он есть сейчас, мы можем, например, вызвать функцию print_hello() с аргументом, и компилятор даже не заподозрит неладное, потому что на уровне исходного кода нет **четких правил**, регламентирующих взаимодействие клиента и сервера. После того как мы создадим заголовочный файл и включим его в файл main.c, компилятор будет "сматывать удочки" каждый раз, когда мы будем пытаться вызвать функцию print_hello() не по правилам. Таким образом интерфейс (набор объявлений, в данном случае - в заголовочном файле) - это публичная оферта сервера клиенту. Включение заголовочного файла директивой #include - это акцепт или подпись.

Еще хочу сказать пару слов о стандартной библиотеке языка C. Как я уже говорил, библиотека - это набор объектных файлов, которые подсоединяются к программе на стадии линковки. Так как стандартная библиотека языка C - это часть стандарта языка C, то она подключается автоматически во время линковки программы, но так как компилятор gcc сам вызывает линковщик с нужными параметрами, то мы этого просто не замечаем. Включая в исходники заголовочный файл stdio.h, мы автоматически соглашаемся использовать механизмы стандартного ввода-вывода на условиях сервера (стандартной библиотеки языка C).

Теперь попробуем применить модель КИС на практике для нашего проекта Hello World. Создадим файл hello.h:

```
/* hello.h */
void print_hello (void);
```

Теперь включим этот файл в main.c:

```
/* main.c */
#include "hello.h"

int main (void)
{
        print_hello ();
}
```

Так как в проект был добавлен новый файл, надо сделать полную пересборку:

```
$ make clean
rm -f *.o hello
$ make
gcc -c main.c
gcc -c hello.c
gcc -o hello main.o hello.o
$ ./hello
Hello World
Goodbye World
$
```