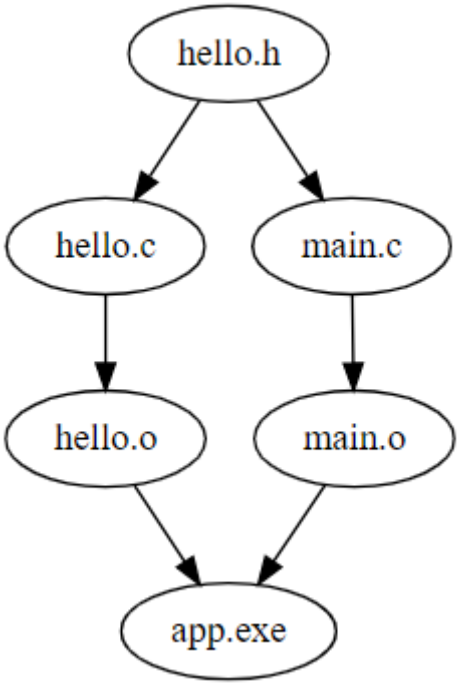
Для сборки многофайлового проекта “Hello”, который был рассмотрен ранее, использовались следующие команды

```
gcc -std=c99 -Wall -Werror -c main.c
gcc -std=c99 -Wall -Werror -c hello.c
gcc -o app.exe hello.o main.o
```

Граф зависимостей для этого проекта выглядит следующим образом



Если меняется какой-то из файлов проекта, то проанализировав граф зависимостей, можно понять какие из команд сборки проекта нужно выполнить, а какие можно опустить. Например, если изменился файл main.c, то нужно получить новый объектный файл main.o и выполнить компоновку приложения. Если же изменяется заголовочный файл hello.h, то приложение придется пересобирать полностью (т.е. выполнить всего команды его сборки). Пока проект состоит из небольшого количества файлов такой анализ можно выполнить вручную. Но если в проекте несколько десятков файлов и некоторые их этих файлов зависят от других файлов, такой анализ становится затруднительным и хочется его автоматизировать. Эту задачу решает утилита make.

make - утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Разновидности утилиты make

- GNU Make (рассматривается далее);
- BSD Make;
- Microsoft Make (nmake).

make выполняет команды согласно правилам, указанным в специальном файле. Этот файл называется *сценарием сборки проекта* или *make-файлом*. Сценарий сборки описывает отношения между файлами программы (граф зависимостей) и содержит команды для обновления каждого файла. Утилита make умеет автоматически определять какая часть

программы изменилась и пересобирает только ее. Для этого она использует время последнего изменения каждого файла и информацию из сценария сборки.

make предполагает, что по умолчанию сценарий сборки *makefile* или *Makefile*. Если это не так, имя make-файла можно указать с помощью ключа **"-f"**:

```
make -f hello-make
```

1. Сценарий сборки

Сценарий сборки состоит из *правил* и *переменных*.

Правила имеют следующий синтаксис:

```
цель: зависимость_1 зависимость_2 … зависимость_n
[tab] команда_1
[tab] команда_2
…
[tab] команда_m
```

Обычно имя цели совпадает с именем файла, который будет создан в результате выполнения команд. Однако, кроме обычных целей существуют так называемые ложные цели (англ. phony targets), которые используются для выполнения некоторых действий (например, очистки). Имена этих целей не являются именами файлов.

Зависимости – это файлы, на основе которых создается цель. Зависимости могут отсутствовать.

Команды – это действия, которые выполняет утилита make для создания цели. Команда будет выполняться всякий раз, когда изменится хотя бы одна из зависимостей. Команды могут отсутствовать.

Обратите внимание, что команды обязательно начинаются с символа табуляции.

Правило без команд говорит только о наличии зависимостей. Правило без зависимостей говорит только о том, какие действия нужно выполнить, но не о том когда это нужно сделать.

Замечание
У утилиты make есть так называемые «встроенные правила» (англ. implicit rules, их описание можно посмотреть с помощью команды "make -qp"), с помощью которых эта утилита может самостоятельно выполнять некоторые действия (например, получить объектный файл).

2. Простой сценарий сборки

makefile

```
 1. app.exe: main.o hello.o
 2.       gcc -o app.exe main.o hello.o
 3.
 4. main.o: main.c hello.h
 5.       gcc -std=c99 -Wall -Werror -c main.c
 6.
 7. hello.o: hello.c hello.h
 8.       gcc -std=c99 -Wall -Werror -c hello.c
10.
11. clean:
```

```
12.        rm *.o *.exe
```

Как вы видите, в сценарии сборки записаны ровно те команды, которые использовались ранее для получения исполняемого файла. Эти команды дополнены информацией о зависимостях между файлами проекта (см. граф зависимостей).

Цель clean используется для удаления файлов, которые были получены в результате предыдущей сборки приложения. Эта цель описывает только действия, поэтому у нее нет зависимостей.

Для сборки приложения в папке, в которой расположен исходный код программы и сценарий сборки, достаточно выдать команду

```
$ make

gcc -std=c99 -Wall -Werror -c main.c
gcc -std=c99 -Wall -Werror -c hello.c
gcc -o app.exe main.o hello.o
```

Если при запуске утилиты make в командной строке не указана цель, утилита make будет обрабатывать первую цель (в нашем случае это app.exe). Эта цель называется *целью по умолчанию*.

Ниже показан запуск утилиты make для выполнения цели clean

```
$ make clean

rm *.o *.exe
```

3. Как make обрабатывает сценарий сборки

Как уже говорилось, если специально не указано, утилита make начинает обработку сценария сборки с цели по умолчанию. В нашем случае это цель app.exe, которая отвечает за создание исполняемого файла. Для достижения этой цели нужно выполнить соответствующее правило.

Все правила обрабатываются одинаково. Сначала анализируются зависимости, перечисленные в правиле. Затем принимается решение нужно ли выполнять команды, указанные в правиле. Команды выполняются если файл-цель не существует или хотя бы один из файлов-зависимостей изменен позже файла-цели.

Зависимости, указанные в правиле, могут оказаться целями каких-то других правил. В этом случае они обрабатываются так, как было описано выше. Зависимости, которые не являются целями, считаются именами файлов. Если такой файл отсутствует, make выдает ошибку.

Достичь цель app.exe make не может, потому что она зависит от двух других файлов main.o и hello.o. Утилита make находит правило для достижения цели main.o. Эта цель зависит от файлов main.c и hello.h, которые есть, а файл main.o еще не создан. Поэтому выполняется команда для создания файла main.o.

После аналогичной обработки цели hello.o утилита make вернется к цели app.exe и, поскольку все зависимости уже существуют, а файл-цель нет, то выполнит команду компоновки, указанную в правиле.

4. Использование переменных и комментариев

Строки, которые начинаются с символа **"#"**, являются комментариями.

Определить переменную в сценарии сборки можно, например, следующим образом:

```
$OBJECTS=hello.o main.o
```

Чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '**"$"**.

```
$(OBJECTS)
```

Переменные широко используются в сценариях сборки. Например, это удобный способ учесть возможность того, что проект будут собирать другим компилятором или с другими опциями.

makefile_2

```
 1. # Компилятор
 2. CC = gcc
 3.
 4. # Опции компиляции
 5. CFLAGS = -std=c99 -Wall -Werror
 6.
 7. app.exe: main.o hello.o
 8.       $(CC) -o app.exe main.o hello.o
10.
11. main.o: main.c hello.h
12.       $(CC) $(CFLAGS) -c main.c
13.
14. hello.o: hello.c hello.h
15.       $(CC) $(CFLAGS) -c hello.c
16.
17. clean:
18.       $(RM) *.o *.exe
```

Замечание
«Встроенные правила», которые упоминались выше, используются встроенные переменные (англ. implicit variables, их описание также можно посмотреть с помощью команды "make -qp"). К таким «встроенным переменным» относятся переменные CC и CFLAGS, значение которых мы переопределили, и переменная RM, для которой мы используем значение по умолчанию.

5. Автоматические переменные

Автоматические переменные – это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд. Автоматические переменные можно использовать для «упрощения» записи правил.

Переменная	Значение
$@	Заменяется на текущую цель.
$<	Заменяется на первую зависимость из списка.
@^	Заменяется на список всех зависимостей с их каталогами.

makefile_3

```
 1. # Компилятор
 2. CC = gcc
```

```
 3.
 4.  # Опции компиляции
 5.  CFLAGS = -std=c99 -Wall -Werror
 6.
 7.  app.exe: main.o hello.o
 8.        $(CC) -o $@ $^
10.
11.  main.o: main.c hello.h
12.        $(CC) $(CFLAGS) -c $<
13.
14.  hello.o: hello.c hello.h
15.        $(CC) $(CFLAGS) -c $<
16.
17.  clean:
18.        $(RM) *.o *.exe
```

6. Полезные ключи утилиты make

Ключ	Описание
`-B`	Задает безусловное построение. Другими словами, make рассматривает все цели как устаревшие.
`-n`	Выводит команды, которые были бы выполнены, но в действительности их не выполняет.
`-r`	Отключает встроенные неявные правила.
`-f имя_файла`	Задает имя файла сценария сборки.