ЗНАКОМСТВО СО СТАНДАРТНОЙ УТИЛИТОЙ *GNU MAKE* ДЛЯ ПОСТРОЕНИЯ ПРОЕКТОВ В ОС *UNIX*

ЦЕЛЬ РАБОТЫ

Ознакомиться с техникой компиляции программ на языке программирования *C* (*C++*) в среде ОС семейства *Unix*, а также получить практические навыки использования утилиты *GNU make* для сборки проекта.

ЗАДАНИЕ

Изучить особенности работы с утилитой *make* при создании проекта на языке *C* (*C++*) в ОС *Unix*, а также получить практические навыки использования утилиты *GNU make* при создании и сборке проекта.

2.1. ОСНОВЫ ИСПОЛЬЗОВАНИЯ УТИЛИТЫ ПОСТРОЕНИЯ ПРОЕКТОВ *MAKE*

«Сборка» большинства программ для ОС семейства *Unix* производится с использованием утилиты *make*. Эта утилита считывает файл (обычно носящий имя «*makefile*» или «*makefile*»; далее, упоминая имя этого файла, будем использовать *makefile*), в котором содержатся инструкции, и выполняет в соответствии с ними действия, необходимые для сборки программы. Во многих случаях *makefile* полностью генерируется специальной программой. Например, для разработки процедур сборки используются программы *autoconf/automake*. Однако в некоторых программах может потребоваться непосредственное создание файла *makefile* без использования процедур автоматической генерации.

Следует отметить, что существует, как минимум, три различных наиболее распространенных варианта утилиты *make*: *GNU make*, *System V make* и *Berkeley make*.

2.1.1. Основные правила работы с утилитой *make*

Основными составляющими любого *make*-файла являются правила (*rules*). В общем виде правило выглядит так:

<цель_1> ... <цель_n>: <зависимость_1> ... <зависимость_n>

<команда_1>

...

<команда_n>

Цель (*target*) – это некоторый желаемый результат, способ достижения которого описан в правиле. Цель может представлять собой имя файла. В этом случае правило описывает, каким образом можно получить новую версию этого файла.

В следующем примере целью является файл *iEdit* (исполняемый файл программы некоторого гипотетического проекта текстового редактора с главным файлом проекта *main.cpp* и дополнительными – *Editor.cpp, TextLine.cpp*). Правило описывает, каким образом можно получить новую версию бинарного файла *iEdit* (скомпоновать из перечисленных объектных файлов):

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
```

Если необходимо скомпилировать проект, написанный на *C++*, то можно использовать компилятор *g++*. Следует также отметить, что ключ *o* компилятора *gcc* указывает имя конечно бинарного файла.

Цель также может быть именем некоторого действия. В таком случае правило описывает, каким образом совершается указанное действие. В следующем примере целью является действие *clean* (очистка, удаление):

```
clean:
rm *.o iEdit
```

Подобного рода цели называют псевдоцелями (*pseudo targets*) или абстрактными целями (*phony targets*).

Зависимость (*dependency*) – это некие «исходные данные», необходимые для достижения указанной в правиле цели, некоторое «предварительное условие» для достижения цели. Зависимость может представлять собой имя файла. Для того чтобы успешно достичь указанной цели, этот файл должен существовать.

В следующем правиле файлы *main.o*, *Editor.o* и *TextLine.o* являются зависимостями. Эти файлы должны существовать для того, чтобы стало возможным достижение цели – построение файла *iEdit*:

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
```

Зависимость также может быть именем некоторого действия. Это действие должно быть предварительно выполнено перед достижением цели, указанной в правиле. В следующем примере зависимость *clean_obj* является именем действия (удалить объектные файлы программы):

```
clean_all:  clean_obj
rm iEdit
clean_obj:
rm *.o
```

Для того, чтобы достичь цели *clean_all*, необходимо сначала выполнить действие (достигнуть цели) *clean_obj*.

Команды – это действия, которые необходимо выполнить для обновления либо достижения цели. В следующем примере командой является вызов компилятора *gcc*. Утилита *make* отличает строки, содержащие команды, от прочих строк *make*-файла по наличию символа табуляции (символа с кодом 9) в начале строки:

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
```

В приведенном выше примере строка *gcc main.o Editor.o TextLine.o -o iEdit* должна начинаться с символа табуляции.

Общий алгоритм работы *make*

Типичный *make*-файл проекта содержит несколько правил. Каждое из правил имеет некоторую цель и некоторые зависимости. Смыслом работы *make* является достижение цели, которую она выбрала в качестве главной цели (*default goal*). Если главная цель является именем действия (т. е. абстрактной целью), то смысл работы *make* заключается в выполнении соответствующего действия. Если же главная цель является именем файла, то программа *make* должна построить самую «свежую» версию указанного файла.

Выбор главной цели. Главная цель может быть прямо указана в командной строке при запуске *make*. В следующем примере *make* будет стремиться достичь цели *iEdit* (получить новую версию файла *iEdit*)

```
make iEdit
```

В этом примере *make* должна достичь цели *clean* (очистить директорию от объектных файлов проекта)

```
make clean
```

Если не указывать какой-либо цели в командной строке, то *make* выбирает в качестве главной первую встреченную в *make*-файле цель. В следующем примере из четырех перечисленных в *make*-файле целей (*iEdit*, *main.o*, *Editor.o*, *TextLine.o*, *clean*) по умолчанию в качестве главной будет выбрана цель *iEdit*:

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
main.o: main.cpp
gcc -c main.cpp
Editor.o: Editor.cpp
gcc -c Editor.cpp
TextLine.o: TextLine.cpp
gcc -c TextLine.cpp
clean:
rm *.o
```

Схематично «верхний уровень» алгоритма работы *make* можно представить так:

```
make()
{
    главная_цель = ВыбратьГлавнуюЦель ()
    ДостичьЦели (главная_цель)
}
```

Достижение цели. После того как главная цель выбрана, *make* запускает «стандартную» процедуру достижения цели. Сначала в *make*-файле выполняется поиск правила, которое описывает способ достижения этой цели (функция «НайтиПравило»). Затем к найденному правилу применяется обычный алгоритм обработки правил (функция «ОбработатьПравило»):

```
ДостичьЦели (Цель)
{
    правило = НайтиПравило (Цель)
    ОбработатьПравило (правило)
}
```

Обработка правил. Обработка правила разделяется на два основных этапа. На первом этапе обрабатываются все зависимости, перечисленные в правиле (функция «ОбработатьЗависимости»). На втором этапе принимается решение о том, нужно ли выполнять указанные в правиле команды (функция «НужноВыполнятьКоманды»). При необходимости перечисленные в правиле команды выполняются (функция «ВыполнитьКоманды»):

```
ОбработатьПравило(Правило)
{
    ОбработатьЗависимости (Правило)
    если НужноВыполнятьКоманды (Правило)
      {
      ВыполнитьКоманды (Правило)
      }
}
```

Обработка зависимостей. Функция «ОбработатьЗависимости» поочередно проверяет все перечисленные в правиле зависимости. Некоторые из них могут оказаться целями каких-нибудь правил. Для этих зависимостей выполняется обычная процедура достижения цели (функция «ДостичьЦели»). Те зависимости, которые не являются целями, считаются именами файлов. Для таких файлов проверяется факт их наличия. При их отсутствии *make* аварийно завершает работу с сообщением об ошибке.

```
ОбработатьЗависимости (Правило)
{
        цикл от i=1 до Правило.число_зависимостей
          {
          если ЕстьТакаяЦель (Правило.зависимость[ i ])
            {
            ДостичьЦели (Правило.зависимость[ i ])
            }
          иначе
            {
            ПроверитьНаличиеФайла (Правило.зависимость[ i ])
            }
          }
    }
```

Обработка команд. На стадии обработки команд решается вопрос о том, следует ли выполнять описанные в правиле команды или нет. Считается, что нужно выполнять команды в таких случаях, как:

- цель является именем действия (абстрактной целью);
- цель является именем файла и этого файла не существует;

- какая-либо из зависимостей является абстрактной целью;
- цель является именем файла и какая-либо из зависимостей, являющихся именем файла, имеет более позднее время модификации, чем цель.

В противном случае (т. е. ни одно из вышеприведенных условий не выполняется) описанные в правиле команды не выполняются. Алгоритм принятия решения о выполнении команд схематично можно представить так:

```
НужноВыполнятьКоманды (Правило)
{
    если Правило.Цель.ЯвляетсяАбстрактной ()
       return  true
    //  цель является именем файла
    если ФайлНеСуществует (Правило.Цель)
       return  true
    цикл от i=1 до Правило.Число_зависимостей
       {
     если Правило.Зависимость[ i ].ЯвляетсяАбстрактной ()
          retum  true
       иначе
            //  зависимость является именем файла
            {
         если ВремяМодификации(Правило.Зависимость[ i ]) >
                ВремяМодификации (Правило.Цель)
              return  true
            }
       }
     return  false
}
```

Абстрактные цели и имена файлов. Имена действий от имен файлов утилита *make* отличает следующим образом: сначала выполняется поиск файла с указанным именем, и если файл найден, то считается что цель или зависимость являются именем файла; в противном случае считается, что данное имя является либо именем несуществующего файла, либо именем действия (различия между этими двумя вариантами не делается, поскольку они обрабатываются одинаково).

Следует отметить, что подобный подход имеет ряд недостатков. Во-первых, утилита *make* не рационально расходует время, выполняя поиск несуществующих имен файлов, которые на самом деле являются именами действий. Во-вторых, при подобном подходе имена действий не должны совпадать с именами каких-либо файлов или директорий, иначе *make*-файл будет выполняться ошибочно.

Некоторые версии *make* предлагают свои варианты решения этой проблемы. Так, например, в утилите *GNU make* имеется механизм (специальная цель *.PHONY*), с помощью которого можно указать, что данное имя является именем действия.

2.1.2. Пример практического использования утилиты *make*

Пример создания простейшего *make*-файла

Рассмотрим, как утилита *make* будет обрабатывать такой *make*-файл (*makefile*):

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
main.o: main.cpp
gcc -c main.cpp
Editor.o: Editor.cpp
gcc -c Editor.cpp

TextLine.o: TextLine.cpp
gcc -c TextLine.cpp
clean:
rm *.o
```

Предположим, что в директории с проектом находятся следующие файлы:

```
main.cpp
Editor.cpp
TextLine.cpp
```

Предположим также, что программа *make* была вызвана следующим образом:

```
make
```

Цель не указана в командной строке, поэтому запускается алгоритм выбора цели (функция «ВыбратьГлавнуюЦель»). Главной целью становится файл *iEdit* (первая цель из первого правила). Цель *iEdit* передается

функции «ДостичьЦели». Эта функция выполняет поиск правила, которое описывает обрабатываемую цель. В данном случае это первое правило *make*-файла. Для найденного правила запускается процедура обработки (функция «ОбработатьПравило»).

Сначала поочередно обрабатываются описанные в правиле зависимости (функция «ОбработатьЗависимости»). Первая зависимость – объектный файл *main.o*. Поскольку в *make*-файле есть правило с такой целью (функция «ЕстьТакаяЦель» возвращает *true*), то для цели *main.o* запускается процедура «ДостичьЦели».

Функция «ДостичьЦели» ищет правило, где описана цель *main.o*. Эта цель описана во втором правиле *make*-файла. Для этого правила запускается функция «ОбработатьПравило», которая запускает процесс обработки зависимостей (функция «ОбработатьЗависимости»). Во втором правиле указана единственная зависимость – *main.cpp*. Такой цели в *make*-файле не существует, поэтому считается, что зависимость *main.cpp* является именем файла. Далее, проверяется наличие этого файла на диске (функция «ПроверитьНаличиеФайла») – такой файл существует. На этом процесс обработки зависимостей завершается.

После обработки зависимостей функция «ОбработатьПравило» принимает решение о том, следует ли выполнять указанные в правиле команды (функция «НужноВыполнятьКоманды»). Цели правила (файла *main.o*) не существует, поэтому команды выполнять следует. Функция «ВыполнитьКоманды» запускает указанную в правиле команду (компилятор *gcc*), в результате чего создается файл *main.o*, так называемый объектный файл (*object file*).

Цель *main.o* достигнута (объектный файл *main.o* построен). Теперь make возвращается к обработке остальных зависимостей первого правила. Зависимости *Editor.o* и *TextLine.o* обрабатываются аналогично. Для них выполняются те же действия, что и для зависимости *main.o*.

После того, как все зависимости (*main.o*, *Editor.o* и *TextLine.o*) обработаны, решается вопрос о необходимости выполнения указанных в правиле команд (функция «НужноВыполнятьКоманды»).

Поскольку цель (*iEdit*) является именем файла, который в данный момент не существует, то принимается решение выполнить описанную в правиле команду (функция «ВыполнитьКоманды»).

Содержащаяся в правиле команда запускает компилятор *gcc*, в результате чего создается исполняемый (бинарный) файл *iEdit*. Таким образом, главная цель (*iEdit*) достигнута. На этом программа *make* завершает свою работу.

Рассмотрим еще один пример работы утилиты *make* в условиях, когда для обработки описанного выше *make*-файла будет выполнена команда

```
make clean
```

Цель явно указана в командной строке, поэтому главной целью становится абстрактная цель *clean*. Цель *clean* передается функции «ДостичьЦели». Эта функция ищет правило, которое описывает обрабатываемую цель. Это будет пятое правило *make*-файла. Для найденного правила запускается процедура обработки (функция «ОбработатьПравило»).

Поскольку в правиле не указано каких-либо зависимостей, *make* сразу переходит к этапу обработки указанных в правиле команд. Цель является именем действия, поэтому команды нужно выполнять. Указанные в правиле команды выполняются, и цель *clean*, таким образом, считается достигнутой. На этом программа *make* завершает работу.

Использование переменных

Возможность использования переменных внутри *make*-файла – очень удобное и часто используемое свойство *make*. В традиционных версиях утилиты, переменные ведут себя подобно макросам языка *C*. Для задания значения переменной используется оператор присваивания. Например, выражение

```
obj_list := main.o Editor.o TextLine.o
```

присваивает переменной *obj_list* значение «*main.o Editor.o TextLine.o*» (без кавычек). Пробелы между символом «=» и началом первого слова игнорируются. Следующие за последним словом пробелы также. Значение переменной можно использовать с помощью конструкции

```
$(имя_переменной)
```

Например, при обработке такого *make*-файла

```
dir_list := . .. src/include
all:
echo $(dir_list)
```

на экран будет выведена строка

```
. .. src/include
```

Переменные могут не только содержать текстовые строки, но и «ссылаться» на другие переменные. Например, в результате обработки *make*-файла

```
optimize_flags := -O3
compile_flags  := $(optimize_flags) -pipe
all:
echo $(compile_flags)
```

на экран будет выведено

```
-O3 -pipe
```

Во многих случаях использование переменных позволяет упростить *make*-файл и повысить его наглядность. Для того чтобы облегчить модификацию *make*-файла, можно разместить «ключевые» имена и списки в отдельных переменных и поместить их в начало *make*-файла:

```
program_name := iEdit
obj_list     := main.o Editor.o TextLine.o
$(program_name): $(obj_list)
gcc $(obj_list) -o $(program_name)
...
```

Адаптация такого *make*-файла для сборки другой программы сведется к изменению нескольких начальных строк.

Использование автоматических переменных

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. Такое, например, правило

```
iEdit: main.o Editor.o TextLine.o
gcc  main.o Editor.o TextLine.o -o iEdit
```

с использованием автоматических переменных можно записать следующим образом:

```
iEdit: main.o Editor.o TextLine.o
gcc  $^  -o $@
```

Здесь *$^* и *$@* являются автоматическими переменными. Переменная *$^* означает «список зависимостей». В данном случае при вызове компилятора *gcc* она будет ссылаться на строку «*main.o Editor.o TextLine.o*». Переменная *$@* означает «имя цели» и будет в этом примере ссылаться на имя «*iEdit*». Если бы в примере была использована следующая автоматическая переменная *$<*, то она указывала бы на первое имя зависимости, т. е. в данном случае на файл *main.o*.

Иногда использование автоматических переменных совершенно необходимо, например в шаблонных правилах.

Шаблонные правила

Шаблонные правила (*implicit rules* или *pattern rules*) – это правила, которые могут быть применены к целой группе файлов. В этом их отличие от обычных правил, описывающих отношения между конкретными файлами. Традиционные реализации *make* поддерживают так называемую «суффиксную» форму записи шаблонных правил:

```
.<расширение_файлов_зависимостей>.<расширение_файлов_целей>:
    <команда_1>
    <команда_2>
    ...
    <команда_n>
```

Например, следующее правило гласит, что все файлы с расширением «*o*» зависят от соответствующих файлов с расширением «*cpp*»:

```
.cpp.o:
gcc -c $^
```

Для современной реализации *make* более предпочтительная следующая запись данной цели:

```
%.o:  %.cpp
gcc -c $^ -o $@
```

Следует обратить внимание на использование автоматической переменной *$^* для передачи компилятору имени файла-зависимости. Поскольку шаблонное правило может применяться к разным файлам, использование автоматических переменных – это единственный способ узнать для каких файлов задействуется правило в данный момент. Шаблонные правила позволяют упростить *make*-файл и сделать его более универсальным. Рассмотрим простой проектный файл:

```
iEdit: main.o Editor.o TextLine.o
gcc $^ -o $@
main.o: main.cpp
gcc -c $^
Editor.o: Editor.cpp
gcc -c $^
TextLine.o: TextLine.cpp
gcc -c $^
```

Все исходные тексты программы обрабатываются одинаково: для них вызывается компилятор *gcc*. С использованием шаблонных правил этот пример можно переписать так:

```
iEdit: main.o Editor.o TextLine.o
gcc $^ -o $@
%.o: %.cpp
gcc -c $^
```

Когда *make* ищет в файле проекта правило, описывающее способ достижения искомой цели (см. п. «Достижение цели», функция «НайтиПравило»), то в расчет принимаются и шаблонные правила. Для каждого из них проверяется, нельзя ли задействовать это правило для достижения искомой цели.

Пример создания более сложного *make*-файла

Предыдущие два примера создания *make*-файлов существенно упрощают создание проектов. Следует отметить, что работа по перечислению всех объектных файлов, составляющих программу, может быть также автоматизирована. При этом вариант создания бинарного файла типа

```
iEdit: *.o
gcc $< -o $@
```

может не сработать, т. к. в указанном случае будут учтены только существующие в данный момент объектные файлы. Особенно это актуально в случае наличия сложных заголовочных файлов (**.h*), определющих зависимости частей проекта. Для того чтобы избежать подобного затруднения, следует использовать более сложный способ, который основан на предположении, что все файлы, содержащие исходный текст, должны быть скомпилированы и скомпонованы в результирующую программу. Такой вариант методики сборки состоит из двух шагов:

1. Получить список всех файлов с исходным текстом программы (всех файлов с расширением «*cpp*»). Для этого следует использовать функции обработки строк, в данном случае функцию *wildcard*, которая получает список файлов с заданным шаблоном в указанном каталоге.

2. Преобразовать список исходных файлов в список объектных файлов (заменить расширение «*cpp*» на расширение объектных файлов «*o*»). Для этого следует воспользоваться функцией *patsubst*, которая заменяет заданную подстроку в заданной строке.

Следующий пример содержит модифицированную версию *make*-файла с использованием указанных двух шагов:

```
iEdit: $(patsubst  %.cpp,%.o,$(wildcard  *.cpp))
gcc $^ -o $@
%.o: %.cpp
gcc -c $<
main.o: main.h Editor.h TextLine.h
Editor.o: Editor.h TextLine.h
TextLine.o: TextLine.h
```

Список объектных файлов программы строится автоматически (цель *iEdit*). Сначала с помощью функции *wildcard* получается список всех файлов с расширением «*cpp*», находящихся в директории проекта. Затем с помощью функции *patsubst* полученный таким образом список исходных файлов преобразуется в список объектных файлов (расширение файлов меняется с «*cpp*» на «*o*»). *Make*-файл теперь стал более универсальным.

Автоматическое построение зависимостей от заголовочных файлов

Автоматизировав перечисление объектных файлов, остается проблема перечисления зависимостей объектных файлов от заголовочных файлов. Например:

```
....
main.o: main.h Editor.h TextLine.h
Editor.o: Editor.h TextLine.h
TextLine.o: TextLine.h
```

Перечисление зависимостей «вручную» может потребовать существенного объема работы. Не всегда достаточно просто открыть файл с исходным текстом и перечислить имена всех заголовочных файлов, подключаемых с помощью директивы «*#include*»: заголовочные файлы могут включать в себя другие заголовочные файлы и подобная «цепочка зависимостей» может быть достаточно длинной. Построение списка зависимостей можно реализовать с использованием утилиты *make* и компилятора *gcc*.

Для совместной работы с *make* компилятор *gcc* имеет несколько опций:

- Ключ компиляции *M*. Для каждого файла с исходным текстом препроцессор будет выдавать на стандартный выход список зависимостей в виде правила для программы *make*. В список зависимостей попадает сам исходный файл, а также все файлы, включаемые с помощью директив «*#include <имя_файла>*» и «*#include "имя_файла"*». После за-

пуска препроцессора компилятор останавливает работу и генерации объектных файлов не происходит.

- Ключ компиляции *MM*. Аналогичен ключу *M*, но в список зависимостей попадает только сам исходный файл и файлы, включаемые с помощью директивы «*#include "имя_файла"*».
- Ключ компиляции *MD*. Аналогичен ключу *M*, но список зависимостей выдается не на стандартный выход, а записывается в отдельный файл зависимостей. Имя этого файла формируется из имени исходного файла путем замены его расширения на расширение «*d*». Например, файл зависимостей для файла *main.cpp* будет называться *main.d*. В отличие от ключа *M* компиляция проходит обычным образом, а не прерывается после фазы запуска препроцессора.
- Ключ компиляции *MMD*. Аналогичен ключу -*MD*, но в список зависимостей попадает только сам исходный файл, и файлы, включаемые с помощью директивы «*#include "имя_файла"*».

Как видно, компилятор может работать двумя способами – в одном случае он выдает только список зависимостей и заканчивает работу (опции *M* и *MM*). В другом случае компиляция происходит как обычно, только в дополнении к объектному файлу генерируется еще и файл зависимостей (опции *MD* и *MMD*). Предпочтительней использовать второй вариант, т. к.

1) при изменении какого-либо из исходных файлов будет построен заново лишь один соответствующий ему файл зависимостей;

2) построение файлов зависимостей происходит «параллельно» с основной работой компилятора и практически не отражается на времени компиляции.

Из двух возможных опций, *MD* и *MMD*, предпочтительней первая, т. к. с помощью директивы «*#include <имя_файла>*» часто включаются не только «стандартные» (например «*#include <iostream.h>*»), но и свои собственные заголовочные файлы, которые могут иногда меняться (например, «*#include «myclass.h*»).

После того как файлы зависимостей сформированы, они имеют расширение «*d*». Для того, чтобы сделать их доступными утилите *make*, следует использовать директиву *#include*:

```
include $(wildcard *.d)
```

Следует обратить внимание на использование функции *wildcard*, т. к. конструкция

```
include *.d
```

будет правильно работать только в том случае, если в каталоге будет находиться хотя бы один файл с расширением «*d*». Если таких файлов

нет, то *make* аварийно завершится, т. к. потерпит неудачу при попытке «построить» эти файлы. Если же использовать функцию *wildcard*, то при отсутствии искомых файлов эта функция просто вернет пустую строку. Далее, директива *include* с аргументом в виде пустой строки, будет проигнорирована, не вызывая ошибки. Теперь новый вариант *makefile* из этого примера выглядит следующим образом:

```
iEdit: $(patsubst %.cpp,%.o,$(wildcard *.cpp))
gcc $^ -o $@
%.o: %.cpp
gcc -c -MD $<
include $(wildcard *.d)
```

Файлы с расширением «*d*» – это сгенерированные компилятором *gcc* файлы зависимостей. Вот, например, как выглядит файл *Editor.d*, в котором перечислены зависимости для файла *Editor.cpp*:

```
Editor.o: Editor.cpp Editor.h TextLine.h
```

Теперь при изменении любого из файлов, *Editor.cpp, Editor.h* или *TextLine.h*, файл *Editor.cpp* будет перекомпилирован для получения новой версии файла *Editor.o*.

Размещение файлов с исходными текстами по директориям

Приведенный выше *make*-файл вполне работоспособен и с успехом может быть использован для сборки небольших программ. Однако с увеличением размера программы, становится не очень удобным хранить все файлы с исходными текстами в одном каталоге. В таком случае предпочтительно размещать эти файлы по разным директориям, отражающим логическую структуру проекта. Для этого нужно модифицировать *make*-файл, чтобы неявное правило

```
%.o: %.cpp
gcc -c $<
```

осталось работоспособным. Для этого используют переменную *VPATH*, в которой перечисляются все директории, где могут располагаться исходные тексты. В следующем примере файлы *Editor.cpp* и *Editor.h* расположены в каталоге *Editor*, а файлы *TextLine.cpp* и *TextLine.h* в каталоге *TextLine*:

```
main.cpp
main.h
Editor /
```

```
Editor.cpp
Editor.h
TextLine /
TextLine.cpp
TextLine.h
makefile
```

Вот как выглядит *makefile* для этого примера:

```
source_dirs := Editor TextLine
search_wildcard s := $(addsuffix /*.cpp,$(source_dirs))
iEdit: $(notdir $(patsubst %.cpp,%.o,$(wildcard $(search_wildcard s))))
gcc $^ -o $@
VPATH := $(source_dirs)
%.o: %.cpp
gcc -c -MD $(addprefix -I ,$(source_dirs)) $<
include $(wildcard *.d)
```

По сравнению с предыдущим вариантом *make*-файла он претерпел следующие изменения:

- для хранения списка директорий с исходными текстами, который нужно будет указывать в нескольких местах, заведена отдельная переменная *source_dirs*;
- шаблон поиска для функции *wildcard* (переменная *search_wildcard s*) строится «динамически», исходя из списка директорий *source_dirs*;
- переменная *VPATH* используется для поиска файлов с исходными текстами;
- компилятору разрешается искать заголовочные файлы во всех директориях с исходными текстами; для этого используется функция *addprefix*, добавляющая префикс-флаг «*I*» к строке компилятора *gcc*;
- при формировании списка объектных файлов из имен исходных файлов «убирается» имя каталога, где они расположены (с помощью функции *notdir*);
- при формировании списка исходных файлов с расширением «*cpp*» была использована функция *addsuffix*, добавляющая суффикс «/*.*cpp*» к названиям каталогов с исходными файлами, указанными в переменной *source_dirs*.

Сборка программы с разными параметрами компиляции

Часто возникает необходимость в получении нескольких вариантов программы, скомпилированных с использованием различным параметров. Типичным примером использования двух различных вариантов является использование отладочной и рабочей версии программы. В таких случаях следует использовать подход, при котором

1) все варианты программы собираются с помощью одного и того же *make*-файла;

2) необходимые настройки компилятора осуществляются в *make*-файле с использованием параметров, передаваемых программе *make* в командной строке (например, флаги компиляции)

```
make compile_flags="-O3 -funroll-loops -fomit-frame-pointer"
```

Таким образом, наиболее простым способом задания параметров компиляции будет внесение дополнительной переменной *compile_flags* в *makefile*. Если параметров компиляции несколько (строка с параметрами содержит пробелы), то строка со значением переменной *compile_flags* должна быть заключена в кавычки. Командный файл *makefile* с использованием параметров может выглядеть следующим образом:

```
override compile_flags := –pipe
source_dirs := Editor TextLine
search_wildcard s := $(addsuffix /*.cpp,$(source_dirs))
iEdit: $(notdir $(patsubst %.cpp,%.o,$(wildcard $(search_wildcard s))))
gcc $^ $(compile_flags) -o $@
VPATH := $(source_dirs)
%.o: %.cpp
gcc -c -MD $(addprefix -I ,$(source_dirs)) $(compile_flags) $<
include $(wildcard *.d)
```

Переменная *compile_flags* получает свое значение из командной строки, если оно задано, в противном случае используется значение по умолчанию, т. е. «*pipe*». Для ускорения работы компилятора к параметрам компиляции добавляется флажок *pipe*. Следует обратить внимание на необходимость использования в примере директивы *override*, использованной для изменения переменной *compile_flags* внутри *make*-файла. В противном случае переданные флаги компиляции из командной строки не будут размещены в переменной *compile_flags*.

2.2. ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическим материалом.
2. Используя любой текстовый редактор, создать простейшую программу на языке *C* (*C++*) с использованием, как минимум, двух исходных файлов (с программным кодом).
3. Для автоматизации сборки проекта утилитой *make* создать *make*-файл (см. п. «Пример создания более сложного *make*-файла»).
4. Выполнить программу (скомпилировать, при необходимости отладить).
5. Показать, что при изменении одного исходного файла и последующем вызове *make* будут исполнены только необходимые команды компиляции (неизмененные файлы перекомпилированы не будут) и изменены атрибуты и/или размер объектных файлов (файлы с расширением *.o*).
6. Создать *make*-файл с высоким уровнем автоматизированной обработки исходных файлов программы согласно следующим условиям:

- имя скомпилированной программы (выполняемый или бинарный файл), флаги компиляции и имена каталогов с исходными файлами и бинарными файлами (каталоги *src*, *bin* и т. п.) задаются с помощью переменных в *makefile*;
- зависимости исходных файлов на языке *C* (*C++*) и цели в *make*-файле должны формироваться динамически;
- наличие цели *clean*, удалающей временные файлы;
- каталог проекта должен быть структурирован следующим образом:
  - *src* – каталог с исходными файлами;
  - *bin* – каталог с бинарными файлами (скомпилированными);
  - *makefile*.

2.3. ТРЕБОВАНИЯ К ОТЧЕТУ В РАБОТЕ № 2

Отчет должен содержать следующие разделы:

1. Титульный лист, оформленный согласно утвержденному образцу.
2. Цели.
3. Задание.
4. Исходные тексты созданных программ, содержимое созданных *make*-файлов, иллюстрацию результатов работы.
5. Выводы.