Версия 1.2

Эффективное использование GNU Make

(C) Владимир Игнатов, 2000

Оглавление

0. Предисловие

В этой книге я описываю свой опыт работы с утилитой **GNU Make** и, в частности, мою методику подготовки make-файлов. Я считаю свою методику довольно удобной, поскольку она предполагает:

- Автоматическое построение списка файлов с исходными текстами
- Автоматическую генерацию зависимостей от включаемых файлов (с помощью компилятора **GCC**)
- "Параллельную" сборку отладочной и рабочей версий программы

Моя книга построена несколько необычным образом. Как правило, книги строятся по принципу "от простого - к сложному". Для новичков это удобно, но может вызвать затруднение у профессионалов. Опытный программист будет вынужден "продираться" сквозь книгу, пропуская главы с известной ему информацией. Я решил построить книгу по другому принципу. Вся "квинтэссенция" книги, ее "главная идея", содержится в первой главе. Остальные главы носят более или менее дополнительный характер.

В начале каждой главы я кратко описываю, о чем в ней будет вестись речь, и какими знаниями нужно обладать, чтобы успешно воспринять излагаемый в главе материал. Для тех, кто чувствует, что недостаточно хорошо ориентируется в предмете разговора, я указываю на дополнительные главы, с которыми следует предварительно ознакомиться.

Для работы я использовал **GNU Make** версии 3.79.1. Некоторые старые версии **GNU Make** (например версия 3.76.1 из дистрибутива **Slackware 3.5**) могут неправильно работать с примером "традиционного" строения make-файла (по-видимому, они "не воспринимают" старую форму записи шаблонных правил).

1. Моя методика использования GNU Make

В этой главе я описываю свой способ построения make-файлов для сборки проектов с использование программы **GNU Make** и компилятора **GCC** *(GNU Compiler Collection)* . Предполагается, что вы хорошо знакомы с утилитой **GNU Make**. Если это не так, то прочтите сначала главу 2 - "***GNU Make***" .

1.1. Пример проекта

В качестве примера я буду использовать "гипотический" проект - текстовой редактор. Он состоит из нескольких файлов с исходным текстом на языке **C++** (*main.cpp*, *Editor.cpp*, *TextLine.cpp*) и нескольких включаемых файлов (*main.h*,*Editor.h*, *TextLine.h*). Если вы имеете доступ в интернет то "электронный" вариант приводимых в книге примеров можно получить на моей домашней страничке по адресу *www.geocities.com/SiliconValley/Office/6533* . Если интернет для вас недоступен, то в Приложении *D* приведены листинги файлов, которые используются в примерах.

1.2. "Традиционный" способ построения make-файлов

В первом примере make-файл построен "традиционным" способом. Все исходные файлы собираемой программы находятся в одном каталоге:

- example_1-traditional /
  - *main.cpp*
  - *main.h*
  - *Editor.cpp*
  - *Editor.h*

- *TextLine.cpp*
- *TextLine.h*
- *Makefile*

Предполагается, что для компиляции программы используется компилятор **GCC**, и объектные файлы имеют расширение "*.o*". Файл *Makefile* выглядит так:

```
#
#   example_1-traditional/Makefile
#
#   Пример "традиционного" строения make-файла
#

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

.cpp.o:
    gcc -c $<

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

Первое правило заставляет **make** перекомпоновывать программу при изменении любого из объектных файлов. Второе правило говорит о том, что объектные файлы зависят от соответствующих исходных файлов. Каждое изменение файла с исходным текстом будет вызывать его перекомпиляцию. Следующие несколько правил указывают, от каких заголовочных файлов зависит каждый из объектных файлов. Такой способ построения make-файла мне кажется неудобным потому что:

- Требуется "явно" перечислять все объектные файлы, из которых компонуется программа
- Требуется "явно" перечислять, от каких именно заголовочных файлов зависит тот или иной объектный файл
- Исполняемый файл программы помещается в "текущую" директорию. Если мне нужно иметь несколько различных вариантов программы (например, отладочный и рабочий), то каждый раз при переходе от одного варианта к другому требуется полная перекомпиляция программы во избежание нежелательного "смешивания" разных версий объектных файлов.

Видно, что традиционный способ построения make-файлов далек от идеала. Единственно чем этот способ может быть удобен - своей "совместимостью". По-видимому, с таким make-файлом будут нормально работать даже самые "древние" или "экзотические" версии **make** (например, **nmake** фирмы **Microsoft**). Если подобная "совместимость" не нужна, то можно сильно облегчить себе жизнь, воспользовавшись широкими возможностями утилиты **GNU Make**. Попробуем избавиться от недостатков "традиционного" подхода.

1.3. Автоматическое построение списка объектных файлов

"Ручное" перечисление всех объектных файлов, входящих в программу - достаточно нудная работа, которая, к счастью, может быть автоматизирована. Разумеется "простой трюк" вроде:

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

не сработает, так как будут учтены только *существующие* в данный момент объектные файлы. Я использую чуть более сложный способ, который основан на предположении, что

все файлы с исходным текстом должны быть скомпилированы и скомпонованы в собираемую программу. Моя методика состоит из двух шагов:

- Получить список всех файлов с исходным текстом программы (всех файлов с расширением *cpp*). Для этого можно использовать функцию **wildcard**.
- Преобразовать список исходных файлов в список объектных файлов (заменить расширение "*.cpp*" на расширение "*.o*"). Для этого можно воспользоваться функцией **patsubst**.

Следующий пример содержит модифицированную версию make-файла:

- example_2-auto_obj /
    - *main.cpp*
    - *main.h*
    - *Editor.cpp*
    - *Editor.h*
    - *TextLine.cpp*
    - *TextLine.h*
    - *Makefile*

Файл *Makefile* теперь выглядит так:

```
#
#   example_2-auto_obj/Makefile
#
#   Пример автоматического построения списка объектных файлов
#

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. 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. Make-файл теперь стал более универсальным - с небольшими изменениями его можно использовать для сборки разных программ.

1.4. Автоматическое построение зависимостей от заголовочных файлов

"Ручное" перечисления зависимостей объектных файлов от заголовочных файлов - занятие еще более утомительное и неприятное чем "ручное" перечисление объектных файлов. Указывать такие зависимости обязательно нужно - в процессе разработки программы заголовочные файлы могут меняться довольно часто (описания классов, например, традиционно размещаются в заголовочных файлах). Если не указывать зависимости объектных файлов от соответствующих заголовочных файлов, то может сложиться ситуация, когда разные объектные файлы программы будут скомпилированы с использованием разных версии одного и того же заголовочного файла. А это, в свою очередь, может привести к частичной или полной потере работоспособности собранной программы.

Перечисление зависимостей "вручную" требует довольно кропотливой работы. Недостаточно

просто открыть файл с исходным текстом и перечислить имена всех заголовочных файлов, подключаемых с помощью *#include*. Дело в том, что одни заголовочные файлы могут, в свою очередь, включать в себя другие заголовочные файлы, так что придется отслеживать всю "цепочку" зависимостей.

Утилита **GNU Make** не сможет самостоятельно построить список зависимостей, поскольку для этого придется "заглядывать" внутрь файлов с исходным текстом - а это, разумеется, лежит уже за пределами ее "компетенции". К счастью, трудоемкий процесс построения зависимостей можно автоматизировать, если воспользоваться помощью компилятора **GCC**. Для совместной работы с **make** компилятор **GCC** имеет несколько опций:

Ключ компиляции	Назначение
-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 запуска препроцессора компилятор останавливает работу, и генерации объектных файлов не происходит.
-MM	Аналогичен ключу *-M*, но в список зависимостей попадает только сам исходный файл, и файлы, включаемые с помощью директивы *#include "имя_файла"*
-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 "*.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.
-MMD	Аналогичен ключу *-MD*, но в список зависимостей попадает только сам исходный файл, и файлы, включаемые с помощью директивы *#include "имя_файла"*

Как видно из таблицы компилятор может работать двумя способами - в одном случае компилятор 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*). Я предпочитаю использовать второй вариант - он мне кажется более удобным и экономичным потому что:

- При изменении какого-либо из исходных файлов будет построен заново лишь один соответствующий ему файл зависимостей
- Построение файлов зависимостей происходит "параллельно" с основной работой компилятора и практически не отражается на времени компиляции

Из двух возможных опций *-MD* и *-MMD*, я предпочитаю первую потому что:

- С помощью директивы *#include <имя_файла>* я часто включаю не только "стандартные", но и свои собственные заголовочные файлы которые могут иногда меняться (например, заголовочные файлы моей прикладной библиотеки *LIB*).
- Иногда бывает полезно взглянуть на *полный* список включаемых в модуль заголовочных файлов, в том числе и "стандартных".

После того как файлы зависимостей сформированы, нужно сделать их доступными утилите **make**. Этого можно добиться с помощью директивы **include**.

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

Обратите внимание на использование функции **wildcard**. 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, так как потерпит неудачу при попытке "построить" эти файлы (у нее ведь нет на этот счет ни каких инструкций!). 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 можно составить новый вариант make-файла для моего "гипотического" проекта:

- example_3-auto_depend /
    - *main.cpp*
    - *main.h*
    - *Editor.cpp*
    - *Editor.h*
    - *TextLine.cpp*
    - *TextLine.h*
    - *Makefile*

Вот как выглядит *Makefile* из этого примера:

```
#
#   example_3-auto_depend/Makefile
#
#   Пример автоматического построения зависимостей от заголовочных файлов
#

iEdit: $(patsubst %.cpp,%.o,$(wildcard *.cpp))
    gcc $^ -o $@

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

include $(wildcard *.d)
```

После завершения работы **make** директория проекта будет выглядеть так:

- example_3-auto_depend /
    - *iEdit*
    - *main.cpp*
    - *main.h*
    - *main.o*
    - *main.d*
    - *Editor.cpp*
    - *Editor.o*
    - *Editor.d*
    - *Editor.h*
    - *TextLine.cpp*
    - *TextLine.o*
    - *TextLine.d*
    - *TextLine.h*
    - *Makefile*

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*.

Имеет ли описанная методика недостатки? Да, к сожалению, имеется один недостаток. К счастью, на мой взгляд, не слишком существенный. Дело в том что утилита **make** обрабатывает make-файл "в два приема". Сначала будет обработана директива **include** и в make-файл будут включены файлы зависимостей, а затем, на "втором проходе", будут уже выполняться необходимые действия для сборки проекта.

Получается что для "текущей" сборки используются файлы зависимостей, сгенерированные во время "предыдущей" сборки. Как правило, это не вызывает проблем. Сложности возникнут лишь в том случае, если какой-нибудь из заголовочных файлом по какой-либо причине прекратил свое существование. Рассмотрим простой пример. Предположим, у меня имеются файлы *main.cpp* и *main.h*:

Файл *main.cpp*:

```
#include "main.h"

void main()
{
}
```

Файл *main.h*:

```
// main.h
```

В таком случае, сформированный компилятором файл зависимостей *main.d* будет выглядеть так:

```
main.o: main.cpp main.h
```

Теперь, если я переименую файл *main.h* в *main_2.h*, и соответствующим образом изменю файл *main.cpp*,

Файл *main.cpp*:

```
#include "main_2.h"

void main()
{
}
```

то очередная сборка проекта окончится неудачей, поскольку файл зависимостей *main.d* будет ссылаться на не существующий более заголовочный файл *main.h*.

Выходом в этой ситуации может служить удаление файла зависимостей *main.d*. Тогда сборка проекта пройдет нормально и будет создана новая версия этого файла, ссылающаяся уже на заголовочный файл *main_2.h*:

```
main.o: main.cpp main_2.h
```

При переименовании или удалении какого-нибудь "популярного" заголовочного файла, можно просто заново пересобрать проект, удалив предварительно все объектные файлы и файлы зависимостей.

1.5. "Разнесение" файлов с исходными текстами по директориям

Приведенный в предыдущем параграфе 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 я предпочитаю "разносить" их по разным директориям, отражающим логическую структуру проекта. Для этого нужно немного модифицировать make-файл. Чтобы неявное правило

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

осталось работоспособным, я использую переменную **VPATH**, в которой перечисляются все директории, где могут располагаться исходные тексты. В следующем примере я поместил файлы *Editor.cpp* и *Editor.h* в каталог *Editor*, а файлы *TextLine.cpp* и *TextLine.h* в каталог *TextLine*:

- example_4-multidir /
    - *main.cpp*
    - *main.h*
    - Editor /
        - *Editor.cpp*
        - *Editor.h*
    - TextLine /
        - *TextLine.cpp*
        - *TextLine.h*
    - *Makefile*

Вот как выглядит *Makefile* для этого примера:

```
#
#   example_4-multidir/Makefile
#
#   Пример "разнесения" исходных текстов по разным директориям
#

source_dirs := . Editor TextLine

search_wildcards := $(addsuffix /*.cpp,$(source_dirs))

iEdit: $(notdir $(patsubst %.cpp,%.o,$(wildcard $(search_wildcards))))
    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 я завел отдельную переменную *source_dirs*, поскольку этот список понадобится указывать в нескольких местах.
- Шаблон поиска для функции **wildcard** (переменная *search_wildcards*) строится "динамически" исходя из списка директорий *source_dirs*
- Используется переменная **VPATH** для того, чтобы шаблонное правило могло искать файлы исходных текстов в указанном списке директорий

- Компилятору разрешается искать заголовочные файлы во всех директориях с исходными текстами. Для этого используется функция **addprefix** и флажок *-I* компилятора **GCC**.
- При формировании списка объектных файлов, из имен исходных файлов "убирается" имя каталога, где они расположены (с помощью функции **notdir**)

1.6. Сборка программы с разными параметрами компиляции

Часто возникает необходимость в получении нескольких вариантов программы, которые были скомпилированы по-разному. Типичный пример - отладочная и рабочая версии программы. В таких случаях я использую простую методику:

- Все варианты программы собираются с помощью одного и того же make-файла.
- Необходимые настройки компилятора "попадают" в make-файл через параметры, передаваемые программе **make** в командной строке.

Для каждой конфигурации программы я делаю маленький командный файл, который вызывает **make** с нужными параметрами:

- example_5-multiconfig /
    - *main.cpp*
    - *main.h*
    - Editor /
        - *Editor.cpp*
        - *Editor.h*
    - TextLine /
        - *TextLine.cpp*
        - *TextLine.h*
    - *Makefile*
    - *make_debug*
    - *make_release*

Файлы *make_debug* и *make_release* - это командные файлы, используемые для сборки соответственно отладочной и рабочей версий программы. Вот, например, как выглядит командный файл *make_release*:

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

Обратите внимание, что строка со значением переменной *compile_flags* заключена в кавычки, так как она содержит пробелы. Командный файл *make_debug* выглядит аналогично:

```
make  compile_flags="-O0 -g"
```

Вот как выглядит *Makefile* для этого примера:

```
#
#   example_5-multiconfig/Makefile
#
#   Пример получения нескольких версий программы с помощью одного make-файла
#

source_dirs := . Editor TextLine

search_wildcards       := $(addsuffix /*.cpp,$(source_dirs))
override compile_flags += -pipe

iEdit: $(notdir $(patsubst %.cpp,%.o,$(wildcard $(search_wildcards))))
    gcc $^ -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 и, далее, используется при компиляции исходных текстов. Для ускорения работы компилятора, к параметрам компиляции добавляется флажок **-pipe**. Обратите внимание на необходимость использования директивы **override** для изменения переменной *compile_flags* внутри make-файла.

1.7. "Разнесение" разных версий программы по отдельным директориям

В том случае если я собираю несколько вариантов одной и той же программы (например, отладочную и рабочую версию), становится неудобным помещать результаты компиляции в один и тот же каталог. При переходе от одного варианта к другому приходится полностью перекомпилировать программу во избежание нежелательного "смешивания" объектных файлов разных версий.

Для решения этой проблемы я помещаю результаты компиляции каждой версии программы в свой отдельный каталог. Так, например, отладочная версия программы (включая все объектные файлы) помещается в каталог *debug*, а рабочая версия программы - в каталог *release*:

- example_6-multiconfig-multidir /
    - debug /
    - release /
    - *main.cpp*
    - *main.h*
    - Editor /
        - *Editor.cpp*
        - *Editor.h*
    - TextLine /
        - *TextLine.cpp*
        - *TextLine.h*
    - *Makefile*
    - *make_debug*
    - *make_release*

Главная сложность заключалась в том, чтобы заставить программу **make** помещать результаты работы в разные директории. Попробовав разные варианты, я пришел к выводу, что самый легкий путь - использование флажка *--directory* при вызове **make**. Этот флажок заставляет утилиту перед началом обработки make-файла, сделать каталог, указанный в командной строке, "текущим".

Вот, например, как выглядит командный файл *make_release*, собирающий рабочую версию программы (результаты компиляции помещается в каталог *release*):

```
mkdir  release
make    compile_flags="-O3 -funroll-loops -fomit-frame-pointer" \
      --directory=release \
      --makefile=../Makefile
```

Команда *mkdir* введена для удобства - если удалить каталог *release*, то при следующей сборке он будет создан заново. В случае "составного" имени каталога (например, *bin/release*) можно

дополнительно использовать флажок *-p*. Флажок *--directory* заставляет **make** перед началом работы сделать указанную директорию *release* текущей. Флажок *--makefile* укажет программе **make**, где находится make-файл проекта. По отношению к "текущей" директории *release*, он будет располагаться в "родительском" каталоге.

Командный файл для сборки отладочного варианта программы (*make_debug*) выглядит аналогично. Различие только в имени директории, куда помещаются результаты компиляции (*debug*) и другом наборе флагов компиляции:

```
mkdir   debug
make    compile_flags="-O0 -g" \
      --directory=debug \
      --makefile=../Makefile
```

Вот окончательная версия make-файла для сборки "гипотического" проекта текстового редактора:

```
#
#   example_6-multiconfig-multidir/Makefile
#
#   Пример "разнесения" разных версий программы по отдельным директориям
#

program_name := iEdit
source_dirs  := . Editor TextLine

source_dirs      := $(addprefix ../,$(source_dirs))
search_wildcards := $(addsuffix /*.cpp,$(source_dirs))

$(program_name): $(notdir $(patsubst %.cpp,%.o, $(wildcard $
(search_wildcards) ) ) )
        gcc $^ -o $@

VPATH := $(source_dirs)

%.o: %.cpp
    gcc -c -MD $(compile_flags) $(addprefix -I,$(source_dirs)) $<

include $(wildcard *.d)
```

В этом окончательном варианте я "вынес" имя исполняемого файла программы в отдельную переменную *program_name*. Теперь для того чтобы адаптировать этот make-файл для сборки другой программы, в нем достаточно изменить всего лишь несколько первых строк.

После запуска командных файлов *make_debug* и *make_release* директория с последним примером выглядит так:

- example_6-multiconfig-multidir /
    - debug /
        - *iEdit*
        - *main.o*
        - *main.d*
        - *Editor.o*
        - *Editor.d*
        - *TextLine.o*
        - *TextLine.d*
    - release /
        - *iEdit*
        - *main.o*
        - *main.d*

        - *Editor.o*
        - *Editor.d*
        - *TextLine.o*
        - *TextLine.d*
    - *main.cpp*
    - *main.h*
    - Editor /
        - *Editor.cpp*
        - *Editor.h*
    - TextLine /
        - *TextLine.cpp*
        - *TextLine.h*
    - *makefile*
    - *make_debug*
    - *make_release*

Видно, что объектные файлы для рабочей и отладочной конфигурации программы помещаются в разные директории. Туда же попадают готовые исполняемые файлы и файлы зависимостей.

В этой главе я изложил свою методику работы с make-файлами. Остальные главы носят более или менее "дополнительный" характер.

- В *Приложении A* я описываю проблемы, которые могут возникнуть при редактировании make-файлов в разных операционных системах
- В *Приложении B* я описываю свой личный способ организации дерева каталогов для сложных проектов.
- В *Приложении C* я делюсь некоторыми мыслями по поводу использования компилятора **GCC**

2. GNU Make

В этой главе я кратко опишу некоторые возможности программы **GNU Make**, которыми я пользуюсь при написании своих make-файлов, а также укажу на ее отличия от "традиционных" версий **make**. Предполагается, что вы знакомы с принципом работы подобных программ. В противном случае сначала прочтите глава 3 - *Утилита **make*** .

GNU Make - это версия программы **make** распространяемая **Фондом Свободного Программного Обеспечения** (**Free Software Foundation - FSF**) в рамках проекта **GNU** (*www.gnu.org*). Получить самую свежую версию программы и документации можно на "домашней страничке" программы *www.gnu.org/software/make* либо на страничке **Paul D. Smith** - одного из авторов **GNU Make** (*www.paulandlesley.org/gmake*).

Программа **GNU Make** имеет очень подробную и хорошо написанную документацию, с которой я настоятельно рекомендую ознакомиться. Если у вас нет доступа в интернет, то пользуйтесь документацией в формате *Info*, которая должна быть в составе вашего дистрибутива **Linux**. Будьте осторожны с документацией в формате man-странички (*man make*) - как правило, она содержит лишь отрывочную и сильно устаревшую информацию.

2.1. Две разновидности переменных

GNU Make поддерживает два способа задания переменных, которые несколько различаются по смыслу. Первый способ - традиционный, с помощью оператора '=':

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

Такой способ поддерживают все варианты утилиты **make**. Его можно сравнить, например, с заданием макроса в языке **Си**.

```
#define  compile_flags  "-O3 -funroll-loops -fomit-frame-pointer"
```

Значение переменной, заданной с помощью оператора '=', будет вычислено в момент ее использования. Например, при обработке make-файла:

```
var1 = one
var2 = $(var1) two
var1 = three

all:
    @echo $(var2)
```

на экран будет выдана строка "three two". Значение переменной *var2* будет вычислено непосредственно в момент выполнения команды *echo*, и будет представлять собой *текущее* значение переменной *var1*, к которому добавлена строка *" two"*. Как следствие - одна и та же переменная не может одновременно фигурировать в левой и правой части выражения, так как это может привести к бесконечной рекурсии. **GNU Make** распознает подобные ситуации и прерывает обработку make-файла. Следующий пример вызовет ошибку:

```
compile_flags = -pipe $(compile_flags)
```

GNU Make поддерживает также и второй, новый способ задания переменной - с помощью оператора '**:=**':

```
compile_flags := -O3 -funroll-loops -fomit-frame-pointer
```

В этом случае переменная работает подобно "обычным" текстовым переменным в каком-нибудь из языков программирования. Вот приблизительный аналог этого выражения на языке **C++**:

```
string   compile_flags = "-O3 -funroll-loops -fomit-frame-pointer";
```

Значение переменной вычисляется в момент обработки оператора присваивания. Если, например, записать

```
var1 := one
var2 := $(var1) two
var1 := three

all:
    @echo $(var2)
```

то при обработке такого make-файла на экран будет выдана строка "one two".

Переменная может "менять" свое поведение в зависимости от того, какой из операторов присваивания был к ней применен последним. Одна и та же переменная на протяжении своей жизни вполне может вести себя и как "макрос" и как "текстовая переменная".

Все свои make-файлы я пишу с применением оператора '**:=**'. Этот способ кажется мне более удобным и надежным. Вдобавок это более эффективно, так как значение переменной не вычисляется заново каждый раз при ее использовании. Подробнее о двух способах задания переменных можно прочитать в документации на **GNU Make** в разделе *"The Two Flavors of Variables"* .

2.2. Функции манипуляции с текстом

Утилита **GNU Make** содержит большое число полезных функций, манипулирующих

текстовыми строками и именами файлов. В частности в своих make-файлах я использую функции **addprefix**, **addsuffix**, **wildcard**, **notdir** и **patsubst**. Для вызова функций используется синтаксис

```
$(имя_функции  параметр1, параметр2 ... )
```

Функция **addprefix** рассматривает второй параметр как список слов разделенных пробелами. В начало каждого слова она добавляет строку, переданную ей в качестве первого параметра. Например, в результате выполнения make-файла:

```
src_dirs := Editor TextLine
src_dirs := $(addprefix ../../, $(src_dirs))

all:
    @echo $(src_dirs)
```

на экран будет выведено

```
../../Editor ../../TextLine
```

Видно, что к каждому имени директории добавлен префикс "*../../*". Функция **addprefix** обсуждается в разделе "Functions for File Names" руководства по **GNU Make**.

Функция **addsuffix** работает аналогично функции **addprefix**, только добавляет указанную строку в конец каждого слова. Например, в результате выполнения make-файла:

```
source_dirs := Editor  TextLine
search_wildcards := $(addsuffix /*.cpp, $(source_dirs))

all:
    @echo $(search_wildcards)
```

на экран будет выведено

```
Editor/*.cpp  TextLine/*.cpp
```

Видно, что к каждому имени директории добавлен суффикс "*/*.cpp*". Функция **addsuffix** обсуждается в разделе "Functions for File Names" руководства по **GNU Make**.

Функция **wildcard** "расширяет" переданный ей шаблон или несколько шаблонов в список файлов, удовлетворяющих этим шаблонам. Пусть в директории *Editor* находится файл *Editor.cpp*, а в директории *TextLine* - файл *TextLine.cpp*:

- wildcard_example /
  - Editor /
    - *Editor.cpp*
  - TextLine /
    - *TextLine.cpp*
  - *makefile*

Тогда в результате выполнения такого make-файла:

```
search_wildcards := Editor/*.cpp  TextLine/*.cpp
source_files := $(wildcard $(search_wildcards))

all:
    @echo $(source_files)
```

на экран будет выведено

```
Editor/Editor.cpp  TextLine/TextLine.cpp
```

Видно, что шаблоны преобразованы в списки файлов. Функция **wildcard** подробно обсуждается в разделе "The Function **wildcard**" руководства по **GNU Make**.

Функция **notdir** позволяет "убрать" из имени файла имя директории, где он находится. Например, в результате выполнения make-файла:

```
source_files := Editor/Editor.cpp  TextLine/TextLine.cpp
source_files := $(notdir $(source_files))

all:
    @echo $(source_files)
```

на экран будет выведено

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

Видно, что из имен файлов убраны "пути" к этим файлам. Функция **notdir** обсуждается в разделе "Functions for File Names" руководства по **GNU Make**.

Функция **patsubst** позволяет изменить указанным образом слова, подходящие под шаблон. Она принимает три параметра - шаблон, новый вариант слова и исходную строку. Исходная строка рассматривается как список слов, разделенных пробелом. Каждое слово, подходящее под указанный шаблон, заменяется новым вариантом слова. В шаблоне может использоваться специальный символ '%', который означает "любое количество произвольных символов". Если символ '%' встречается в новом варианте слова (втором параметре), то он заменяется текстом, соответствующим символу '%' в шаблоне. Например, в результате выполнения make-файла:

```
source_files := Editor.cpp  TextLine.cpp
object_files := $(patsubst %.cpp, %.o, $(source_files))

all:
    @echo $(object_files)
```

на экран будет выведено

```
Editor.o  TextLine.o
```

Видно, что во всех словах окончание "*.cpp*" заменено на "*.o*". Функция **patsubst** имеет второй, более короткий вариант записи для тех случаев, когда надо изменить суффикс слова (например, заменить расширение в имени файла). Более короткий вариант выглядит так:

```
$(имя_переменной:.старый_суффикс=.новый_суффикс)
```

Применяя "короткий" вариант записи предыдущий пример можно записать так:

```
source_files := Editor.cpp  TextLine.cpp
object_files := $(source_files:.cpp=.o)

all:
    @echo $(object_files)
```

Функция **patsubst** обсуждается в разделе "Functions for String Substitution and Analysis" руководства по **GNU Make**.

2.3. Новый способ задания шаблонных правил

В "традиционных" вариантах **make** шаблонное правило задается с помощью конструкций, наподобие:

```
.cpp.o:
```

```
    gcc $^ -o $@
```

То есть под действие правила попадают файлы с определенными расширениями ("*.cpp*" и "*.o*" в данном случае).

GNU Make поддерживает более универсальный подход - с использованием шаблонов имен файлов. Для задания шаблона используется символ **'%'**, который означает "последовательность любых символов произвольной длины". Символ **'%'** в правой части правила заменяется текстом, который соответствует символу **'%'** в левой части. Пользуясь новой формой записи, приведенный выше пример можно записать так:

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

В своих make-файлах я пользуюсь новой формой записи шаблонных правил, потому что считаю ее более удобной (шаблонные и нешаблонные правила теперь имеют аналогичный синтаксис) и универсальной (можно задавать не только файлы, отличающиеся своими расширениями).

2.4. Переменная VPATH

С помощью переменной **VPATH** можно задать список каталогов, где шаблонные правила будут искать зависимости. В следующем примере:

```
VPATH := Editor TextLine

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

make будет искать файлы с расширением "*.cpp*" сначала в текущем каталоге, а затем, при необходимости, в подкаталогах *Editor* и *TextLine*. Я часто использую подобную возможность, так как предпочитаю располагать исходные тексты в иерархии каталогов, отражающих логическую структуру программы.

Переменная **VPATH** описывается в главе "VPATH: Search Path for All Dependencies" руководства по **GNU Make**. На страничке **Paul D. Smith** есть статья под названием "How Not to Use VPATH" (paulandlesley.org/gmake/vpath.html), в которой обсуждается "неправильный" стиль использования переменной **VPATH**.

2.5. Директива override

Переменные в **GNU Make** могут создаваться и получать свое значение разными способами:

- Задаваться внутри make-файла
- "Автоматически" создаваться программой **make** из переменных среды
- Задаваться через командную строку при вызове программы **make**

Последний случай считается "специальным". Если переменная задана через командную строку, то внутри make-файла нельзя изменить ее значение "обычным" способом. Рассмотрим простой make-файл:

```
compile_flags := -pipe $(compile_flags)

all:
    echo $(compile_flags)
```

Предположим, что переменная *compile_flags* была задана через командную строку при запуске программы **make**:

```
make  compile_flags="-O0 -g"
```

В результате обработки make-файла на экран будет выведена строка:

```
-O0 -g
```

То есть попытка изменить значение переменной *compile_flags* внутри make-файла была проигнорирована. Если все-таки возникает необходимость в изменении переменной, которая была задана с помощью командной строки, нужно использовать директиву **override**. Директива помещается перед именем переменной, которая должна быть изменена:

```
override compile_flags := -pipe $(compile_flags)

all:
    echo $(compile_flags)
```

Теперь в результате обработки make-файла на экран будет выдана строка:

```
-pipe -O0 -g
```

2.6. Добавление текста в строку

Часто возникает необходимость добавить текст к существующей переменной. Для этой цели служит оператор "+=". Добавляемый текст может быть как текстовой константой, так и иметь ссылки на другие переменные:

```
compile_flags += -pipe
compile_flags += $(flags)
```

При использовании этого оператора, "тип" переменной (см. раздел 2.1 "Две разновидности переменных") не меняется - "макросы" остаются "макросами", а "текстовые переменные" по-прежнему остаются таковыми.

Если переменная задана с помощью командной строки, то по-прежнему для изменения ее значения внутри make-файла нужно использовать директиву **override**. В следующем примере предполагается, что переменная *compile_flags* задана в командной строке:

```
override compile_flags += -pipe
override compile_flags += $(flags)
```

2.7. Директива include

С помощью директивы **include** можно включать в обрабатываемый make-файл другие файлы. Работает она аналогично директиве **#include** в языках C и C++. Когда встречается эта директива, обработка "текущего" make-файла приостанавливается и **make** временно "переключается" на обработку указанного в директиве файла. Директива **include** может оказаться полезной для включения в make-файл каких-либо "общих", или автоматически сгенерированных другими программами фрагментов.

В директиве **include** могут быть указаны одно или несколько имен файлов, разделенных пробелами. В качестве имен файлов можно использовать шаблоны:

```
include common.mak

include main.d Editor.d TextLine.d

include *.d
```

Указанные в директиве файлы должны существовать - иначе **make** предпримет попытку

"создать" их, а при невозможности этого достигнуть, выдаст сообщение об ошибке. Директива **include** с пустым списком файлов:

```
include
```

просто игнорируется.

2.8. Автоматические переменные

Программа **GNU Make** поддерживает большое число автоматических переменных. В своих make-файлах я использую следующие автоматические переменные:

Имя автоматической переменной	Значение
$@	Имя цели обрабатываемого правила
$<	Имя первой зависимости обрабатываемого правила
$^	Список всех зависимостей обрабатываемого правила

Полный список автоматических переменных приводится в разделе "Automatic Variables" руководства по **GNU Make**.

2.9. "Комбинирование" правил

В make-файле могут встречаться несколько правил, имеющих одинаковую цель. В таком случае они как бы "комбинируются вместе". Например, следующие два правила:

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

TextLine.o: TextLine.h
```

эквивалентны правилу:

```
TextLine.o: TextLine.cpp TextLine.h
    gcc -c $<
```

Шаблонные и нешаблонные правила также могут "комбинироваться":

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

TextLine.o: TextLine.h
```

Обратите внимание на то, что в обоих пример только в одном из правил указаны исполняемые команды - именно они и будут при необходимости выполняться. При наличии команд в обоих правилах, **make** выдаст предупреждающее сообщение и "в расчет" будут приниматься только команды из последнего правила.

2.10. Make-файл, используемый по умолчанию

Если при вызове программы **GNU Make** не указывать явно, какой make-файл следует обрабатывать, то она пытается найти и обработать файлы *GNUmakefile*, *makefile* и *Makefile* (именно в таком порядке). Руководство по **GNU Make** рекомендует имя *Makefile* для make-файлов, используемых по умолчанию. При "алфавитной" сортировке имен файлов в директории, такое имя будет располагаться ближе к началу списка.

2.11. Специальная цель .PHONY

В традиционных реализациях, у программы **make** нет надежного способа узнать, чем именно является цель, указанная в правиле. Цель может быть как именем действия, так и именем файла. Исходя только из "внешнего вида" правила, различить эти случаи невозможно. Утилита **make** просто ищет на диске файл с именем, которое указано в качестве цели. Если такой файл существует, то цель считается именем файла.

Для целей, которые являются именами действий, такой подход не очень хорош. Во-первых, имя такой цели может случайно совпасть с именем какого-либо файла или директории. И, во-вторых, **make** просто нерационально тратит свое время, занимаясь поиском несуществующих файлов.

В утилите **GNU Make** имеется способ явного объявления целей абстрактными. Для этого используется механизм "специальных целей". *Специальная цель* - это имя, которое имеет специальное значение, когда используется в качестве цели. Для того, например, чтобы объявить перечисленные цели абстрактными, достаточно поместить их в правило со специальной целью **.PHONY**. В следующем примере цель *clean* объявляется абстрактной:

```
.PHONY: clean

clean:
    rm *.o *.d
```

Все возможные специальные цели описаны в главе Special Built-in Target Names" руководства по **GNU Make**.

3. Утилита make

Утилита **make**, входящая в состав практически всех *Unix*-подобных операционных систем - это традиционное средство, применяемое для сборки программных проектов. Она является универсальной программой для решения широкого круга задач, где одни файлы должны автоматически обновляться при изменении других файлов.

При запуске программа **make** читает файл с описанием проекта (make-файл) и, интерпретируя его содержимое, предпринимает необходимые действия. Файл с описанием проекта представляет собой текстовой файл, где описаны отношения между файлами проекта, и действия, которые необходимо выполнить для его сборки.

3.1. Правила

Основным "строительным элементом" make-файла являются *правила* (*rules*). В общем виде правило выглядит так:

```
    <цель_1> <цель_2> ... <цель_n>: <зависимость_1> <зависимость_2> ...
<зависимость_n>
        <команда_1>
        <команда_2>
        ...
        <команда_n>
```

Цель (*target*) - это некий 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:

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

целью является файл *iEdit* (исполняемый файл программы). Правило описывает, каким образом можно получить новую версию файла *iEdit* (скомпоновать из перечисленных объектных файлов).

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).

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

Подобного рода цели называются *псевдоцели* (*pseudotargets*) или *абстрактные цели* (*phony targets*).

Зависимость (*dependency*)- это некие "исходные данные", необходимые для достижения указанной в правиле *цели*. Можно сказать что *зависимость* - это "предварительное условие" для достижения цели. *Зависимость* может представлять собой имя файла. Этот файл должен существовать, для того чтобы можно было достичь указанной цели. В следующем правиле:

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

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

Зависимость также может быть именем некоторого действия. Это действие должно быть предварительно выполнено перед достижением указанной в правиле цели. 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 цель *clean_all* была достигнута, нужно сначала выполнить действие (достигнуть цели) *clean_obj*.

Команды - это действия, которые необходимо выполнить для обновления либо достижения *цели*. В следующем примере:

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

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. В приведенном выше примере строка:

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

должна начинаться с символа табуляции.

3.2. 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 (то есть 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.

3.2.1 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
```

А 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:

```
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
```

из четырех перечисленных в make-файле целей (*iEdit*, *main.o*, *Editor.o*, *TextLine.o*, *clean*) по умолчанию в качестве главной будет выбрана цель *iEdit*. Схематично, "верхний уровень" алгоритма работы **make** можно представить так:

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

    ДостичьЦели( главная_цель )
}
```

3.2.2 Достижение *цели*

После того как *главная цель* выбрана, **make** запускает "стандартную" процедуру достижения цели. Сначала в make-файле ищется правило, 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 )
}
```

3.2.3 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 - 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 )
        }
}
```

3.2.4 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 ] )
            }
        }
}
```

3.2.5 Обработка *команд*

На стадии обработки команд решается вопрос - нужно ли выполнять описанные в правиле команды или нет. Считается, что нужно выполнять команды если:

- 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 (если 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()
            return  true
        иначе
            //  зависимость является именем файла
            {
            если ВремяМодефикации( Правило.Зависимость[ i ] )  >
                 ВремяМодефикации( Правило.Цель )
                return  true
            }
        }

    return  false
}
```

3.3. Абстрактные цели и имена файлов

Каким образом **make** отличает имена действий от имен файлов? Традиционные варианты **make** поступают просто. Сначала ищется файл с таким именем. 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 оба случая обрабатываются одинаково.

Подобный подход не слишком хорош по следующим соображениям. Во-первых, утилита **make** не слишком рационально расходует время, занимаясь поиском 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 подобный алгоритм даст сбой, и make-файл будет работать неправильно.

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.

3.4. Пример работы make

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

```
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 ищет правило, 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*.

Функция *ОбработатьПравило* 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, нужно ли выполнять указанные в правиле команды (функция *НужноВыполнятьКоманды*). Цели правила (файла *main.o*) не существует, поэтому команды нужно выполнять. Функция *ВыполнитьКоманды* запускает указанную в правиле команду (компилятор **GCC**), в результате чего создается файл *main.o*.

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 файл *iEdit*. Главная цель (*iEdit*)таким образом достигнута. На этом программа **make** завершает свою работу.

3.5. Еще один пример работы make

Рассмотрим, как будет действовать утилита **make**, если для обработки описанного в

предыдущей главе make-файла, она будет вызвана следующим образом:

```
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.

3.6. Переменные

Возможность использования переменных внутри make-файла - очень удобное и часто используемое свойство **make**. В традиционных версиях утилиты, переменные ведут себя подобно макросам языка Си. 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 игнорируются. 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.

3.7. 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. 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*".

Иногда использование автоматических переменных совершенно необходимо - например, в шаблонных правилах (о них пойдет речь в следующей главе).

3.8. 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 говорит о том, что все файлы с расширением "*o*" зависят от соответствующих файлов с расширением "*cpp*":

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

Обратите 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 сейчас задействуется правило.

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 $@

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

Когда **make** ищет в файле проекта правило, описывающее способ достижения искомой цели (см. главу 3.2.2. "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.

Приложение A. Редактирование make-файлов в разных операционных системах

Если, наряду с операционной системой **Linux**, вы работаете с операционными системами фирмы **Microsoft** (**DOS**, **Windows**), то при редактировании make-файлов в разных системах могут возникнуть определенные трудности.

Проблема состоит в том, что принятый в *Unix*-подобных операционных системах формат хранения текстовых файлов, несколько отличается от формата **DOS**/**Windows**. В **Unix** каждая строка текстового файла заканчивается символом "перевод строки" (код *0x0A*). В **DOS** и **Windows** текстовые строки разделяются парой символов - "возврат каретки", "перевод строки" (*0x0D*, *0x0A*).

Linux-версия программы **GNU Make** будет нормально работать с make-файлами, написанными в среде **Linux**. Версия утилиты **GNU Make** для **Windows** также будет нормально работать с make-файлами, подготовленными в среде **Windows**. Проблема возникнет лишь в том случае, если попытаться обработать *Linux*-версией программы **GNU Make** текстовой файл, подготовленный в среде **DOS** или **Windows**. Подобная ситуация может возникнуть "нечаянно" - достаточно лишь "сохранить" make-файл для среды **Linux** в текстовом редакторе **DOS**/**Windows**, чтобы он оказался "испорчен".

В отличие от компилятора **GCC**, который просто игнорирует символы "возврат каретки", **GNU Make** рассматривает их как "обычные" символы, которые вполне могут быть частью имени. В результате все слова, находящиеся в конце строк, искажаются, так как сзади к ним добавляется невидимый символ "возврат каретки". В следующем примере имя файла *TextLine.o* будет искажено (его длина будет составлять одиннадцать символов из-за невидимого "возврата строки"):

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

```
	gcc $^ -o $@
```

Разумеется, на диске не найдется файла с таким "странным" именем и make-файл будет работать неверно. Дело осложняется еще и тем, что выдаваемые на экран диагностические сообщения также искажаются (при выводе на экран символ "возврат каретки" возвращает курсор в начало строки) и зачастую представляют собой лишь бессмысленные "обрывки" слов.

Описанную проблему можно решить разными способами. Организационный метод решения - никогда не пытаться редактировать make-файл, находясь в "чужеродной" для него среде. Другой возможный подход - "принудительно" удалять из make-файла символы "возврат каретки" (либо "вручную" - текстовым редактором, либо с помощью подходящей программы).

Приложение B. Организация иерархии каталогов в сложных проектах

Для сложных проектов, состоящих из большого количества файлов, я предпочитаю более сложную организацию каталогов, чем та, которая приводилась в качестве примера в разделе 1.7. "Разнесение разных версий программы по отдельным директориям". Основная идея заключается в том, чтобы файлы с разным "назначением" помещались в разные каталоги. В моих проектах дерево каталогов выглядит примерно так:

- имя_проекта /
  - bin /
    - linux_debug /
    - linux_release /
    - windows_debug /
    - windows_release /
  - doc /
    - *README.txt*
  - project /
    - *Makefile*
    - *make_debug*
    - *make_release*
  - src /

В каталог *bin* помещаются результаты компиляции - объектные и исполняемые файлы, библиотеки и тому подобное. Этот каталог можно "безболезненно" удалить - при следующей компиляции он будет автоматически создан заново. В этом каталоге каждая из возможных конфигураций программы имеет свою отдельную директорию. Как правило, я делаю четыре конфигурации программы - для каждой из двух операционных систем (**Linux** и **Windows**) имеется отладочная и рабочая версии программы.

В директорию *doc* я помещаю различные текстовые файлы - документацию, замечания, список ошибок и тому подобное. Здесь же располагается и файл *README.txt*.

В каталоге *project* находится make-файл проекта и командные файлы, используемые для сборки программы в разных конфигурациях.

В каталог *src* я помещаю исходные тексты программы. Внутри директории *src* имеется своя иерархия каталогов, отражающая логическую структуру программы.

Вот пример make-файла, который работает с подобной структурой директорий проекта:

```
#
#   example_7-complex/project/Makefile
```

```
    #
    #   Пример проекта со "сложной" структурой директорий
    #

    program_name := iEdit
    source_dirs  := . Editor TextLine
    include_dirs := /c/aproj/lib  /c/aproj/lib/linux
    link_flags   := -static

    source_dirs  := $(addprefix ../../src/, $(source_dirs) )
    source_files := $(wildcard $(addsuffix /*.cpp, $(source_dirs) ) )
    object_files := $(notdir $(source_files) )
    object_files := $(object_files:.cpp=.o)

    $(program_name): $(object_files)
        gcc $^ -o $@ $(link_flags) -pipe

    VPATH := $(source_dirs)

    %.o: %.cpp
        gcc $< -c $(compile_flags) $(addprefix -I, $(include_dirs)) $(addprefix
-I, $(source_dirs)) -MD -pipe 2>log

    include $(wildcard *.d)
```

Список директорий, где располагаются файлы с исходными текстами (*source_dirs*), задается относительно каталога *src*. Вот как выглядит командный файл, собирающий отладочную версию программы:

```
mkdir -p ../bin/linux_debug
make  compile_flags="-O0 -g" \
      --directory=../bin/linux_debug \
      --makefile=../../project/Makefile
```

Командный файл, собирающий рабочую версию программы выглядит аналогично:

```
mkdir -p ../bin/linux_release
make  compile_flags="-O3 -funroll-loops -fomit-frame-pointer" \
      --directory=../bin/linux_release \
      --makefile=../../project/Makefile
```

Приложение C. Компилятор GCC

GNU Compiler Collection (**GCC**) - это семейство компиляторов с языков **C**, **C++** и **Object-C**, которые объединены общей технологией и распространяются в рамках проекта **GNU**. Домашняя страничка компилятора находится по адресу www.gnu.org/software/gcc/gcc.html

Этот компилятор является "стандартным" средством для компиляции всех программ, входящих в проект **GNU**. **GCC** также является основным компилятором операционной системы **Linux** - с его помощью компилируется ядро системы.

Версии компилятора

Компилятор GCC развивается весьма динамично - программа улучшается, исправляются обнаруженные ошибки, добавляются новые возможности. Всегда желательно знать с какой версией компиляторы вы в данный момент работаете. Возможно, что эта версия еще не поддерживает нужные вам возможности или содержит ошибки, которые могут повлиять на работоспособность компилируемой программы. Узнать версию компилятора **GCC** можно с помощью ключа **-v**:

```
gcc -v
```

Отладка

"Стандартным" средством для отладки программ, скомпилированных компилятором **GCC**, является отладчик **GDB**. Этот отладчик свободно распространяется в рамках проекта **GNU**. Домашняя страничка отладчика находится по адресу www.gnu.org/software/gdb/gdb.html.

Для подготовки отладочной версии программы с помощью компилятора **GCC**, достаточно отключить оптимизацию и включить генерацию отладочной информации. Для этого я использую следующие опции компиляции:

-g	Генерировать отладочную информацию
-O0	Отключить оптимизацию

Отладчик **GDB** имеет текстовой интерфейс командной строки. Мне этот интерфейс кажется не очень удобными, поэтому я пользуюсь графической оболочкой **DataDisplayDebugger** (**DDD**). Эта оболочка является надстройкой над "текстовыми" отладчиками, реализующей для них удобный графический интерфейс. Программа **DDD** также входит в проект **GNU**. Ее домашняя страничка находится по адресу http://www.gnu.org/software/ddd.
DataDisplayDebugger работает в среде **X-Windows**.

Рабочий вариант

При компиляции рабочего варианта программы, я включаю максимальную оптимизацию по скорости. Возможно это приводит к некоторому увеличению размера программы, но я считаю это не слишком важным. Вряд ли кто-нибудь заметит увеличение размера программы на пять-десять килобайт. В то же время быстродействия программам всегда не хватает.

Компилятор **GCC** имеет большое количество опций, управляющих процессом кодогенерации и оптимизации, с которыми вы можете экспериментировать, добиваясь максимального быстродействия программы. Для своих проектов я использую следующие настройки:

Ключ	Назначение
-O3	Максимальная оптимизация
-fomit-frame-pointer	Не использовать указатель на стековый фрейм. Компилятор будет адресовать переменные в стеке с помощью регистра *ESP* а регистр *EBP* "высвобождается" для использования в качестве регистра общего назначения.
-mcpu=pentium	Оптимизировать код для процессора **Pentium** (однако программа по прежнему будет работать даже на *i386*)

Обработка исключений

Если вы используете механизм *исключений* (*exceptions*) языка **C++**, то при компиляции должна быть включена соответствующая опция:

Ключ компиляции	Назначение
-fexceptions	Включить поддержку механизма исключительных ситуаций языка **C++**

Статическая и динамическая компоновка

По умолчанию компилятор компонует собранную программу с динамическими версиями

стандартных библиотек. Это не всегда удобно. Для того чтобы стандартные библиотеки компоновались статически, нужно использовать опцию **--static**. В этом случае генерируется полностью "статический" код, который для своей работы не требует наличия каких-либо загружаемых библиотек.

Получение листинга

Часто бывает полезным иметь ассемблерный листинг кода, генерируемого компилятором. С помощью такого листинга можно:

- Посмотреть, как те или иные опции оптимизации отражаются на генерируемом коде
- Посмотреть, каким образом компилятор обрабатывает те или иные конструкции языка программирования
- Выявлять ошибки, связанные с неправильной работой кодогенерации в компиляторе
- Узнать, какие в точности опции были включены при компиляции программы

Для получения ассемблерного листинга, я использую следующие опции компилятора **GCC**:

Ключ компиляции	Назначение
-S	Остановиться после стадии компиляции, перед стадией ассемблирования.
-fverbose-asm	Генерировать дополнительные комментарии в ассемблерном листинге. Какие именно "дополнительные комментарии" будут помещены в текст листинга, зависит от версии компилятора.

Обратите внимание на то, что указание флажка **-S** просто "останавливает" компилятор после фазы генерации ассемблерного листинга, то есть процесс компиляции прерывается. Как следствие - процесс сборки программы и процесс генерации ассемблерных листингов "несовместимы" между собой. Можно либо получать листинги, либо собирать программу, но не то и другое одновременно. Для получения листингов я обычно создаю отдельный командный файл, который среди прочих опций компиляции содержит флажки **-S** и **-fverbose-asm**.

Весьма полезная возможность компилятора - помещать в листинг список всех опций компиляции, которые были включены в данный момент. Дело в том, что включение одних опций (например **-O3**) может "автоматически" приводить к включению других опций, а документация к **GCC** не всегда точна в описании подобных зависимостей. Некоторые версии **GCC** всегда помещают в листинг список используемых опций, другие версии делают это только при наличии флажка **-fverbose-asm**.

Переназначение ошибок в файл

По умолчанию, компилятор GCC выдает ошибки в стандартный поток сообщений об ошибках (файл с дескриптором 2). Иногда это не очень удобно - сообщения об ошибках могут быть очень длинными и подробными, а "прокрутка" экрана назад не всегда доступна. Поэтому, я, обычно, перенаправляю сообщения компилятора в файл. Этот файл потом можно просмотреть любым стандартным средством. Вот, например, как может выглядеть правило для компиляции исходных текстов:

```
%.o: %.cpp
    gcc -c $< 2>log
```

Опция -pipe

Компилятор **GCC** обрабатывает программу за несколько проходов, помещая промежуточные

результаты компиляции во временные файлы. Процесс компиляцию можно ускорить, если воспользоваться опцией **-pipe**. При включении этой опции, различные этапы компиляции начинают "общаться" между собой не через временные файлы, а через каналы обмена (*pipes*).

Тексты с символом "возврат каретки"

В отличие от утилиты **GNU Make**, компилятор **GCC** вполне "лояльно" относится к наличию символов "возврат каретки" в компилируемых текстах - такие символы попросту игнорируются. Поэтому, проблем при компиляции исходных текстов, подготовленных в среде **DOS**/**Windows**, возникнуть не должно.

Приложение D. "Гипотический" проект - текстовой редактор

В первой главе "Моя методика использования GNU Make" в качестве примера рассматривается "гипотический" проект - текстовой редактор. Он состоит из трех файлов с исходным текстом на языке **C++** (*main.cpp*, *Editor.cpp* и *TextLine.cpp*), а также трех заголовочных файлов (*main.h*, *Editor.h*, *TextLine.h*). Ниже приведены листинги этих файлов.

Файл *main.cpp*:

```
#include "main.h"

void main()
{
}
```

Файл *main.h*:

```
// main.h
```

Файл *Editor.cpp*:

```
#include "Editor.h"
```

Файл *Editor.h*:

```
#include "TextLine.h"
```

Файл *TextLine.cpp*:

```
#include "TextLine.h"
```

Файл *TextLine.h*:

```
// TextLine.h
```

Я буду рад услышать ваше мнение об этом тексте. Для этого можно воспользоваться гостевой книгой:

Sign Guestbook View Guestbook

На главную страницу