Особенности работы с программой для управления компиляцией.

Цель работы.

Изучить особенности работы с программой управления компиляцией на примере утилиты make.

При работе над комплексными программами, состоящими из большого числа файлов, компиляция всех файлов может занимать длительное время и требует ввода большого числа команд. Утилита make автоматически определяет какие части большой программы должны быть перекомпилированы, и выполняет необходимые для этого действия.

В настоящее время существует множество утилит для отслеживания зависимостей, но make — одна из самых широко распространённых, в первую очередь благодаря тому, что она включена в Unix, начиная с версии PWB/UNIX (англ. Programmer's Workbench), которая содержала инструменты для разработки программного обеспечения.

Существует несколько версий make, основанных на оригинальной make или написанных с нуля, использующих те же самые форматы файлов и базовые принципы и алгоритмы, а также содержащие некоторые улучшения и расширения. Например:

BSD make, основанная на работе Адама де Бура (Adam de Boor) над версией make, с возможностью параллельной сборки; в той или иной форме перешла в FreeBSD, NetBSD и OpenBSD.

GNU make — входит в большинство дистрибутивов GNU/Linux и часто используется в сочетании с GNU build system.

В дальнейшем будем рассматривать GNU make в версии для ОС Windows.

Порядок выполнения работы.

Задание правил преобразования файлов.

Утилита make предназначена для автоматизации преобразования файлов из одной формы в другую. Правила преобразования задаются в скрипте с именем Makefile, который должен находиться в корне рабочей директории проекта. Сам скрипт (иначе называемый make-файл) состоит из набора правил, которые в свою очередь описываются:

1) целями (то, что данное правило делает);

2) реквизитами (то, что необходимо для выполнения правила и получения целей);

3) командами (выполняющими данные преобразования).

В общем виде синтаксис makefile можно представить так:

```
# комментарии начинаются со значка #
```

```
# отступы выполняются исключительно при помощи символов табуляции,
# каждой команде должен предшествовать отступ
<цели>: <зависимости>
	<команда #1>
	...
	<команда #n>
```

То есть, правило make это ответы на три вопроса:

{Что делаем? (цели)} ---> {Из чего делаем? (зависимости)} ---> {Как делаем? (команды)}

В качестве исходных файлов для начала будем использовать те, которые использовались в качестве примеров в первой лабораторной. В самом простом случае, программа будет состоять из одного файла - hworld.c. При этом makefile должен выглядеть так:

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

Перед gcc необходимо указать полный путь к файлу gcc.exe. Для запуска используем следующую команду:

mingw32-make.exe -C <ДИРЕКТОРИЯ>

В качестве директории необходимо указать полный путь к папке с исходными текстами программы и файлом makefile. Здесь и далее используется версия make, входящая в состав IDE Dev-Cpp. Параметр -C указывает путь к директории, где находятся исходные тексты программы и файл makefile.В результате в указанной директории появится файл hello.exe, запустив который можно увидеть результат работы программы.

Теперь рассмотрим вариант с программой из нескольких файлов с исходным текстом: first.c и second.c. Существуют несколько вариантов makefile для такой программы, простейший представлен далее:

```
main: first.c second.c
	C:\dev-cpp\bin\gcc -o main.exe first.c second.c
```

При этом будут скомпилированы оба файла с исходным текстом программы и создан файл main.exe. Данный подход обладает существенным недостатком - в случае если программа состоит из большого числа файлов, при внесении изменений в один из них потребуется перекомпиляция всех для создания исполняемого файла программы. Корректнее разделять компиляцию на этапы трансляции и линковки и запускать только требуемый этап. В этом случае makefile будет выглядеть так:

```
first.o: first.c
	C:\dev-cpp\bin\gcc -c -o first.o  first.c
second.o: second.c
	C:\dev-cpp\bin\gcc -c -o second.o second.c
main: first.o second.o
	C:\dev-cpp\bin\gcc -o main.exe first.o second.o
```

Для создания файла потребуется указать цель, в данном случае main:
mingw32-make.exe main -C <ДИРЕКТОРИЯ>

После запуска make попытается сразу получить цель main, но для ее создания необходимы файлы first.o и second.o, которых пока еще нет. Поэтому выполнение правила будет отложено и make станет искать правила, описывающие получение недостающих реквизитов. Как только все реквизиты будут получены, make вернется к выполнению отложенной цели. Отсюда следует, что make выполняет правила рекурсивно. При повторном вызове make, файлы first.o и second.o будут пересозданы только в случае внесения изменений в файлы с соответствующими исходными текстами.

Использование дополнительных целей.

Помимо целей, описывающих требуемые файлы, существует набор стандартных или фиктивных (phony) целей, среди которых можно выделить следующие:

- all — является стандартной целью по умолчанию. При вызове make ее можно явно не указывать.
- clean — очистить каталог от всех файлов полученных в результате компиляции.
- install — произвести инсталляцию
- uninstall — и деинсталляцию соответственно.

Для того чтобы make не искал файлы с такими именами, их следует определить в Makefile, при помощи директивы .PHONY. Далее показан пример Makefile с целями all и clean.

```
.PHONY: all clean
all: main
clean:
	del *.o
first.o: first.c
```

```
	C:\dev-cpp\bin\gcc -c -o first.o  first.c
second.o: second.c
	C:\dev-cpp\bin\gcc -c -o second.o second.c
main: first.o second.o
	C:\dev-cpp\bin\gcc -o main.exe first.o second.o
```

Следует обратить особое внимание на то, что если файл main уже имеется (остался после предыдущей компиляции) и его реквизиты не были изменены, то команда make ничего не станет пересобирать. Так например, изменив заголовочный файл, случайно не включенный в список реквизитов, можно получить долгие часы головной боли. Поэтому, чтобы гарантированно полностью пересобрать проект, нужно предварительно очистить рабочий каталог:
mingw32-make.exe clean -C <ДИРЕКТОРИЯ>

mingw32-make.exe -C <ДИРЕКТОРИЯ>

Использование переменных.

Чтобы избежать повторения текста в файле makefile и одновременно сделать его более гибким, возможно использование переменных. Переменные в make представляют собой именованные строки и определяются следующим образом:

<VAR_NAME> = <value string>

Существует негласное правило, согласно которому следует именовать переменные в верхнем регистре, например:

SRC = first.c second.c

Так мы определили список исходных файлов. Для использования значения переменной ее следует разыменовать при помощи конструкции $(<VAR_NAME>); например так:

gcc -o hello $(SRC)

Модифицируем представленный ранее makefile с использованием переменных:

```
SOURCES=first.c second.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE=main.exe
FLAGS= -c -o
GCCPATH=C:\dev-cpp\bin\gcc
```

```
.PHONY: all clean
all: $(EXECUTABLE)
clean:
	del *.o
$(OBJECTS): $(SOURCES)
	$(GCCPATH) $(FLAGS) $@ $*.c
$(EXECUTABLE): $(OBJECTS)
	$(GCCPATH) -o $(EXECUTABLE) $(OBJECTS)
```

Теперь разберём написанное. Переменные SOURCES,OBJECTS и EXECUTABLE содержат имена файлов с иходным текстом программы, объектных файлов и исполняемого файла соответственно. Переменная OBJECTS заполняется путём взятия имён файлов из переменной SOURCES и заменой расширения .c на .o. Переменная FLAGS хранит требуемые параметры компилятора. Переменная GCCPATH содержит полный путь к набору компиляторов gcc. При разыменовании используются автоматические переменные $@ и $*. Автоматическая переменная $@ представляет собой текущую цель при использовании списка целей. Автоматическая переменная $* представляет собой текущую цель без расширения.

Задания к лабораторной работе.

Написать makefile для всех программ, написанных в предыдущих лабораторных с возможностью выбора требуемой лабораторной в зависимости от введённой цели. Учесть, что файлы с исходным текстом программы будут лежать в подкаталогах, в которые также должны будут помещены полученные объектные и исполняемые файлы.

Контрольные вопросы.

1) Что такое make и makefile?
2) Какие команды допустимы в makefile?
3) Какие параметры могут быть у программы make?
4) Как работать с переменными в makefile?
5) Может ли makefile иметь имя, отличное от makefile? Если да, то как с ним работать в этом случае.
6) В каком порядке выполняются цели в makefile?

7) Как определяются требуемые для перекомпиляции файлы при внесении изменений в один или несколько файлов с исходными текстами?
8) Какие отличия между BSD make и GNU make?

Рекомендованная литература:

http://www.linuxlib.ru/prog/make_379_manual.html#SEC1
http://www.gnu.org/software/make/manual/make.html
http://www.freebsd.org/cgi/man.cgi?query=make&sektion=1
http://mrbook.org/blog/tutorials/make/
http://rus-linux.net/nlib.php?name=/MyLDP/algol/gnu_make/gnu_make_3-79_russian_manual.html