**Лабораторная работа 5. Знакомство со стандартной утилитой**

**Gnu mаке для построения проектов в ОС Unix/Linux**

**Ознакомиться с техникой компиляции программ на языке про-**

**граммирования С (С++) в среде ОС семейства Unix, а также получить**

**практические навыки использования утилиты GNU таке для сборки**

**проекта.**

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

**Теория:**

2.1 Основы использования утилиты построения проектов маке

«Сборка» большинства программ для ОС семейства Unix производится с использованием утилиты make. Эта утилита считывает файл (обычно носящий имя «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 make и Berkeley make

2.1.1. Основные правила работы с 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 -0 iEdit

Если необходимо скомпилировать проект, написанный на С++, то можно использовать компилятор g++. Следует также отметить, что ключ о 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 \*.о 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 предыдущем примере 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 цели, указанной в правиле. В следующем примере зависимость clean\_obj является именем действия (удалить объектные файлы программы):

clean\_all: clean\_obj

clean\_all: clean\_obj

rm iEdit

clean\_obj:

rm \*.0

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 -0 iEdit

В приведенном выше примере строка gcc main.o Editor.o TextLine.o -0 iEdit должна начинаться с символа табуляции.

2.1.2 Общий алгоритм работы make

Типичный make-файл проекта содержит несколько правил. Каждое из правил имеет некоторую цель и некоторые зависимости. Смыслом работы mаке 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 mаке заключается в выполнении соответствующего действия. Если же главная цель является именем файла, то программа mаке должна построить самую «свежую» версию данного файла.

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

mаке 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 mаке 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 \*.0

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 до Правило.число\_зависимостей

если ЕстьТакаяЦель (Правило.зависимость[ i 1)

ДостичьЦели (Правило.зависимость[ i 1)

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 ()

return 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.

2.1.3. Пример практического использования утилиты make

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 \*.0

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 - обьектный 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 следует выполнять команды. Функция «ВыполнитьКоманды» запускает указанную в правиле команду (компилятор gcc), в результате чего создается файл main.o, так называемый объектный файл (objectfile).

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. о.

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 угилиты, переменные ведут себя подобно макросам языка С. 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)

Ha 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-фаqла:

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 -0 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 «о» зависят от соответствующих файлов с расширением «срр»:

.срр.о:

gcc -с $^

Для современной реализации make более предпочтительная следующая запись данной цели:

%.о: %.срр

gcc -с $^ -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 $л -0 $@

main.o: main.cpp

gcc -с у

Editor.o: Editor.cpp

gcc -с у

TextLine.o: TextLine.cpp

gcc -с у

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 $@

%.о: %.срр

gcc -с $^

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

**Практическая часть:**

1. Ознакомиться с теоретическим материалом.

2. Воспользоваться утилитой make для автоматизации сборки проектов из лабораторной работы №4 «Инструментальные средства разработки Linux»

3. 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 С (С++) 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);

makejile.

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
| Варианта | Структура данных (STL) | Функции обработки (таблица 2) | Тип сортировки  (таблица 3) |
|  | Vector |  |  |
|  | Array |  |  |
|  | Deque |  |  |
|  | List |  |  |
|  | Forward\_List |  |  |
|  | Vector |  |  |
|  | Array |  |  |
|  | Deque |  |  |
|  | List |  |  |
|  | Forward\_List |  |  |
|  | Vector |  |  |
|  | Array |  |  |
|  | Deque |  |  |
|  | List |  |  |
|  | Forward\_List |  |  |
|  | Vector |  |  |
|  | Array |  |  |
|  | Deque |  |  |
|  | List |  |  |
|  | Forward\_List |  |  |
|  | Vector |  |  |
|  | Array |  |  |
|  | Deque |  |  |
|  | List |  |  |
|  | Forward\_List |  |  |

|  |  |
| --- | --- |
| N | Функции обработки структурированных данных |
|  | Сумма отрицательных элементов |
|  | Произведение положительных элементов на четных местах |
|  | Количество элементов меньших заданного числа A |
|  | Определение минимального элемента |
|  | Определение максимального элемента |
|  | Сумма элементов попадающих в промежуток [A,B] |
|  | Произведение отрицательных элементов на нечетных местах |
|  | Количество положительных элементов, на местах кратных 3 |
|  | Определение минимального элемента среди положительных элементов |
|  | Определение максимального среди отрицательных элементов |
|  | Сумма элементов на нечетных местах |
|  | Произведение элементов, больших заданного D |
|  | Количество элементов, попадающих в промежуток [A,B] |
|  | Определение минимального элемента на четных местах |
|  | Определение максимального элемента на нечетных местах |

|  |  |  |
| --- | --- | --- |
| N | Начальная сортировка | Конечная сортировка |
|  | Пузырьковая | Вставками |
|  | Вставками | Выбором минимального |
|  | Обменом | Быстрая |
|  | Шелла | Пузырьковая |
|  | Пузырьковая | Быстрая |
|  | Вставками | Обменом |
|  | Обменом | Выбором максимального элемента |

Все функции размещаются в отдельных файлах.

4. Выполнить программу (скомпилировать, при необходимости отладить) для первого варианта сортировки данных.

5. Изменить тип сортировки и выполнить программу.

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 .о).