Практикум по информатике: 8 факультет, 1 курс, 2 семестр 2011/12 уч. года.
Лабораторная работа №25 по курсам «Языки и методы программирования»/ «Алгоритмы и структуры данных»

Автоматизация сборки программ модульной структуры на языке Си с использованием утилиты make

I. Краткое описание утилиты *make*

Утилита *make* предназначена для упрощения сборки (компиляция, редактирование связей, автоматическая подготовка документации) проектов программ модульной структуры. Характерными особенностями, позволившими этой достаточно простой утилите стать стандартным средством ведения проектов, является её переносимость, легкая настраиваемость на конкретные требования и т. д.

Утилита make очень распространена – ее варианты имеются на всех платформах, где только можно программировать. Например, в состав Linux входит GNU *make,* вместе с Microsoft Visual Studio поставляется утилита *nmake* фирмы Microsoft, вместе с IBM VisualAge – *nmake* фирмы IBM. Использование одной достаточно простой утилиты на всех платформах позволяет унифицировать процесс компиляции и связывания многокомпонентных и многомодульных программ и определяет простой стандарт на распространение программного обеспечения в исходных текстах.

При использовании *make* проект разбивается на программные единицы (чаще всего, на объектные и исполняемые файлы, библиотеки, файлы с исходными текстами программ), между которыми устанавливаются взаимосвязи. Рассмотрим простейший пример. Допустим, есть два модуля, каждый из которых представлен двумя файлами: part1.c и part1.h, part2.c и part2.h (реализация и интерфейс), в которых содержатся функции foo1 и foo2 соответственно. Модуль основной программы prog.c использует обе этих функции. Следовательно, на стадии сборки всех объектных файлов в единый исполняемый модуль редактору связей должен быть указан источник кода для этих подпрограмм.

Сценарий компиляции и редактирования связей такой программы может выглядеть следующим образом:

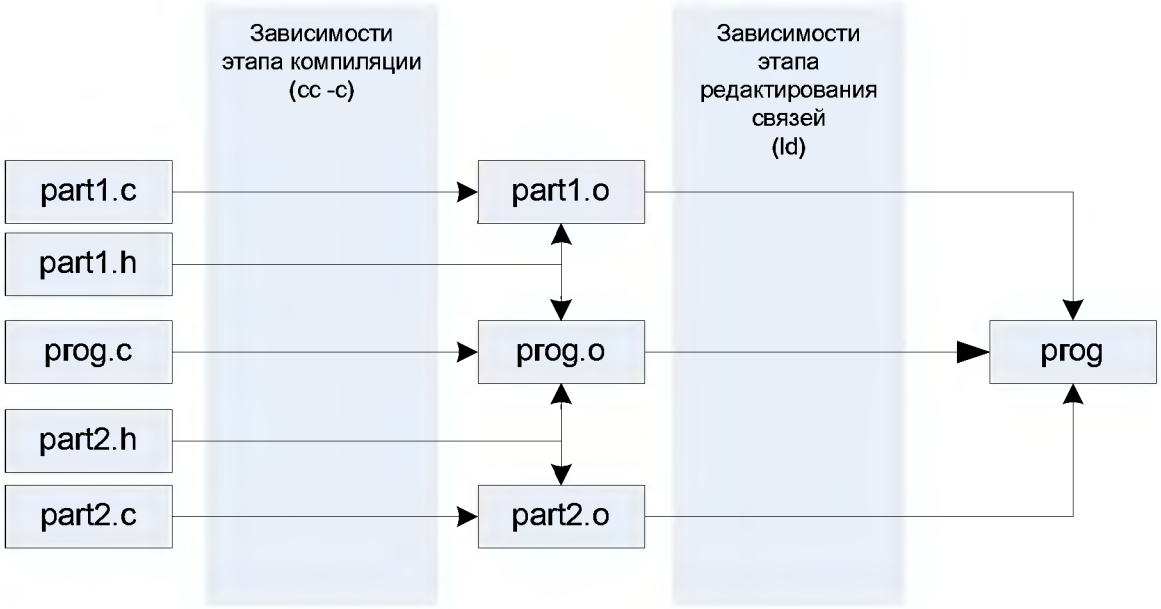
`cc -c part1.c`	компиляция модуля part1.c, содержащего исходный текст для функции foo1
`cc -c part2.c`	компиляция модуля part2.c, содержащего исходный текст для функции foo2
`cc -c prog.c`	компиляция основного модуля программы (содержащего точку входа в модульную программу)
`cc -o prog prog.o part1.o part2.o`	сборка всех модулей в один исполняемый файл prog

Такой подход имеет ряд преимуществ перед обычным вариантом компиляции “в одну строку” командой

```
cc -o prog prog.c part1.c part2.c.
```

Во-первых, в программных проектах, состоящих из многих файлов, при изменении только одного из них нет нужды компилировать все остальные (необходимо перекомпилировать только измененные файлы и отредактировать связи, что может существенно сократить время сборки исполняемого файла). Во-вторых, раздельная компиляция позволяет в многоязычных системах программирования создавать программы, отдельные части которых написаны на разных языках программирования.

Результатом выполнения приведенных выше команд компиляции является файл prog. Можно составить схему процесса компиляции:



Мы будем говорить, что файл A *зависит* от файла B (или нескольких файлов), если файл B необходим для успешного создания файла A с помощью некоторой последовательности команд. В данном примере `part1.o` зависит от `part1.c`, а `prog` зависит от `part1.o`, `part2.o` и `prog.o`, что и отражено на приведенной выше схеме компиляции, которую также можно считать диаграммой зависимости. Кроме того, интерфейсные функции и константы вынесены в заголовочные файлы part1.h и part2.h, которые включаются в текст соответствующих Си-файлов при помощи препроцессора и, следовательно, объектные файлы зависят также и от них.

Для автоматизации процесса сборки выполняемого файла утилитой *make* необходимо составить граф зависимостей для *Makefile* и описать для каждого перехода правила получения «целого по частям» (например, `prog` по объектным файлам). Каждая такая стрелка представляется в *Makefile* таким образом:

```
цель: файл_1 . . . файл_N
        команда_1
              . . .
        команда_m
```

В первой строке записывается вершина графа и список вершин, от которых она зависит. Во всех последующих строках, которые **должны начинаться знаком табуляции**, описываются команды, которые будут поданы интерпретатору команд для того, что бы получить в итоге файл-цель. Таким образом, можно привести примерный вид *makefile* (*makefile* или *Makefile* – стандартные имена файла с правилами для утилиты *make*) для используемого примера:

```
# makefile 1
prog: prog.o part1.o part2.o
        cc -o prog prog.o part1.o part2.o
prog.o: prog.c part1.h part2.h
        cc -c prog.c
part1.o: part1.c part1.h
        cc -c part1.c
part2.o: part2.c part2.h
        cc -c part2.c
```

Все, что следует в строке этого *makefile* за знаком `#`, считается комментарием, и программой *make* игнорируется. Теперь, для того, чтобы по исходным файлам программы создать исполняемый файл `prog`, достаточно запустить утилиту *make* соответствующей командой:

```
bash$ make
```

После этого будут выполнены в указанном порядке все команды, которые необходимы для создания файла prog. Преимуществом использовании этой утилиты является также то, что если после создания исполняемого файла изменить, например, исходный текст подпрограммы foo2 в модуле part2.c, то при компиляции prog утилита make сама определит, какие файлы необходимо перекомпилировать (**все**, *зависимые от измененных!*) и **автоматически** выполнит только те команды, которые затрагивают измененные файлы, т. е. в данном случае компиляцию модуля part2.c и создание исполняемого файла.

Другим важным преимуществом *make* является возможность настройки с помощью переменных (макросов), параметризующих *Makefile*:

```
#  makefile 2
CC = gcc # имя компилятора (cc или gcc)
LD = gcc # имя редактора связей (cc или gcc: редактор связей вызывается
через
         # драйвер системы программирования, когда на вход СП подаются
объектные
         # модули)
CCFLAGS = -g -Wall -pedantic -std=c99 # флаги компилятора (-g -- включить
режим отладочной компиляции)

LDFLAGS =    # флаги связывающего загрузчика (не указаны!)

prog: prog.o part1.o part2.o
         $(LD) $(LDFLAGS) -o prog prog.o part1.o part2.o
prog.o: prog.c  part1.h part2.h
         $(CC) $(CCFLAGS) -c prog.c
part1.o: part1.c part1.h
         $(CC) $(CCFLAGS) -c part1.c
part2.o: part2.c part2.h
         $(CC) $(CCFLAGS) -c part2.c
part1.o: part2.h
```

Последняя строка отражает зависимость первого модуля от интерфейса второго.

Вместо $(*name*) утилита *make* подставит значение переменной с именем *name*, как оно было определено ранее в файле, через командную строку или через т. н. переменные окружения. Теперь одним изменением в *Makefile* можно включить или исключить режим отладки в исполняемых и объектных файлах (установкой нужной комбинации флагов компилятора, т. е. переменной CCFLAGS).

Возможно задание нескольких целей, не зависящих друг от друга, и указание конкретной цели при вызове утилиты *make*. Например, если к предыдущему примеру добавить строки:

```
clean :
         rm -f *.o prog
```

то тем самым будет добавлено еще одно дерево исполнения команд. При выполнении команды make clean будут удалены все объектные файлы и исполняемый файл prog в текущем каталоге.

Если какая-то строка должна быть перенесена (из соображений читабельности), то для переноса используется знак ‘\’, причём за ним должен сразу же следовать конец строки:

```
FILES = \
   part1.c \
   part2.c \
   prog.c
```

Нетрудно заметить, что команды компиляции для part1.c, part2.c и prog.c (также как и для подавляющего большинства Си-программ) однотипны, поэтому утилита *make* позволяет задать правила для файлов с различными суффиксами (чтобы сократить число описаний правил сборки при большом количестве исходных файлов). Недостающие зависимости всё равно приходится прописывать вручную (или использовать ключ компилятора для автоматической генерации зависимостей).

```
# makefile 3
CC = gcc # имя компилятора
LD = gcc # имя редактора связей
CCFLAGS = -g -Wall -pedantic -std=c99 # флаги компилятора (-g --
включить режим отладочной компиляции)

LDFLAGS =    # флаги связывающего загрузчика
OBJ = prog.o part1.o part2.o

.SUFFIXES: .c .o # варианты суффиксов

prog: $(OBJ)

        $(LD) $(LDFLAGS) -o prog $(OBJ)

prog.o: prog.c part1.h part2.h

.c.o :       # компиляция исходных текстов Си
        $(CC) $(CCFLAGS) -c $< -o $@

part1.c: part1.h

part2.c: part2.h


clean:
        rm -f *.o prog
```

Командной `.SUFFIXES` задаются возможные варианты используемых далее суффиксов. Строка для `.c .o` аналогична приведенной ранее для конкретных файлов, за исключением того, что заранее неизвестно, какие будут имена у компилируемых файлов. Пара знаков `$<` обеспечивают подстановку названия файла с исходным текстом для каждого вызова этого правила. (‘<’ – это переменная, в которой содержится имя первого файла во время применения правила с суффиксами). Комбинация $@ обеспечивает подстановку названия выходного файла.

II. Порядок выполнения лабораторной работы.

1. Изучить принципы работы утилиты **make** по книгам или данному руководству.

2. Составить *Makefile* для модульной программы из лабораторной работы №26.

3. Оттестировать *Makefile* и убедиться в его работоспособности для различных ситуаций (изменен один файл программы, два файла, файлы не изменялись).

4. Распечатать протокол с текстом *Makefile* и результат его работы в нескольких различных ситуациях с исходными и объектными файлами (не менее, чем в трёх). Для «изменения» файлов можно воспользоваться утилитой touch (см. оперативную подсказку UNIX). В протокол включить должным образом атрибутированные списки файлов, документирующие состояние файловой системы проекта (***ls –l***).

5. Отчет по лабораторной работе №25 можно оформлять на одном бланке с работой №26.