ЛАБОРАТОРНАЯ РАБОТА № 3.

ПРАКТИЧЕСКОЕ ЗНАКОМСТВО С УТИЛИТОЙ GNU MAKE ДЛЯ СБОРКИ ПРОЕКТОВ В UNIX

ЦЕЛЬ РАБОТЫ

Изучить процесс компиляции программ на языке C (C++) в операционных системах Unix и приобрести практические навыки работы с утилитой GNU make для создания и сборки проектов.

ЗАДАНИЕ

Разобраться в особенностях применения утилиты make при разработке проектов на языке C (C++) в окружении Unix и получить опыт работы с GNU make в процессе создания и компиляции проектов.

2.1. ОСНОВЫ ИСПОЛЬЗОВАНИЯ УТИЛИТЫ MAKE

Большинство программ для Unix-систем собираются с помощью утилиты make. Эта программа анализирует файл (который обычно именуется «makefile») и выполняет действия, прописанные в нем, необходимые для сборки программы. В ряде случаев makefile генерируется автоматически с помощью специальных инструментов, таких как autoconf или automake. Тем не менее, для некоторых проектов может потребоваться создание makefile вручную без автоматизации. Следует также отметить, что существует несколько популярных версий утилиты make: GNU make, System V make и Berkeley make.

2.1.1. Основные принципы работы с утилитой make

Главные элементы любого make-файла — это правила (rules). В общем виде правило записывается так:

<цель_1> … <цель_n>: <зависимость_1> … <зависимость_n>
<команда_1>
…
<команда_n>

Цель (target) — это желаемый результат, путь к которому описан в правиле. Цель может представлять собой имя файла, и в этом случае правило объясняет, как получить обновлённую версию этого файла.

Рассмотрим листинг 1, где целью является файл iEdit (исполняемый файл программы воображаемого текстового редактора с основным файлом main.cpp и дополнительными Editor.cpp, TextLine.cpp). Правило показывает, как создать свежую версию бинарного файла iEdit (компилируя указанные объектные файлы):

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

Листинг 1 — Простейший make файл

Если проект написан на C++, для компиляции используется g++. Важно отметить, что флаг -o компилятора GCC определяет имя выходного бинарного файла.

Цель может также обозначать действие. В листинге 2 целью является действие пол очистке (clean) правило объясняет, как его выполнить. Например, при очистке (clean):

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

Листинг 2 — Цель по очистка проекта

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

З**ависимость (dependency)** — это необходимые «исходные данные» для достижения указанной цели, своего рода «предварительное условие». Это может быть имя файла, который должен существовать для успешного выполнения цели.

В примере листинга 1 объектные файлы main.o, Editor.o и TextLine.o являются зависимостями. Эти файлы нужны, чтобы реализовать цель — создать объектный файл iEdit:

Зависимость может быть и именем действия, которое нужно выполнить перед достижением цели. В листинге 3 - **clean_obj** обозначает действие (удаление объектных файлов):

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

Листинг 3 — Листинг по удалению объектных файлов

Для выполнения clean_all необходимо сначала осуществить действие **clean_obj**.

Команды — это шаги, требуемые для достижения цели. В данном случае командой является вызов компилятора GCC. Утилита make различает команды по символу табуляции в начале строки.

В листинге 1 строка **gcc main.o Editor.o TextLine.o -o iEdit** должна начинаться с символа табуляции (tab).

Алгоритм работы make

Стандартный файл make для проекта включает несколько правил, каждое из которых имеет свою цель и зависимости. Основная функция make заключается в достижении главной цели, определённой как приоритетная (default goal). Если главная цель представляет собой имя действия (то есть абстрактную задачу), то make сосредоточена на выполнении данного действия. Если же главная цель обозначена как имя файла, то программа обязана собрать наиболее актуальную версию указанного файла.

Выбор главной цели.

Главную цель можно задать непосредственно в командной строке при запуске make. Например, команда **make iEdit** заставит make стремиться к достижению цели iEdit, то есть обновлению файла iEdit. Аналогично, команда **make clean** приведёт к выполнению цели clean, которая подразумевает удаление объектных файлов проекта.

При отсутствии конкретной цели в командной строке make выберет первую встреченную цель из файла. В листинге 4 представлен make состоящий из четырёх целей (iEdit, main.o, Editor.o, TextLine.o, clean) в котором будет автоматически выбрана цель iEdit.

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

Листинг 4 — Файл make содержащий четыре цели

Схематически процесс работы make можно изобразить так:

```
make() {
    главная цель = ВыбратьГлавнуюЦель()
    ДостичьЦели(главнаяцель)
}
```

Достижение цели. После выбора основной цели, make инициирует «стандартную» процедуру её реализации. Сначала происходит поиск в make-файле правила, которое определяет метод достижения данной цели (функция «НайтиПравило»). Затем к найденному правилу применяется стандартный алгоритм обработки (функция «ОбработатьПравило»).

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

Обработка правил. Процесс обработки правила делится на два ключевых этапа. На первом этапе производится вытягивание всех зависимостей, перечисленных в правиле (функция «ОбработатьЗависимости»). На втором этапе определяется, необходимо ли выполнять команды, указанные в правиле (функция «НужноВыполнятьКоманды»). Если потребуется, перечисленные команды будут выполнены (функция «ВыполнитьКоманды»).

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

Обработка правил. Обработка правил включает в себя два ключевых этапа. На первом этапе происходит анализ всех зависимостей, прописанных в правиле (функция «ОбработатьЗависимости»). Второй этап посвящен принятию решения о целесообразности выполнения команд, указанных в правиле (функция «НужноВыполнятьКоманды»). При наличии необходимости выполняются команды, упомянутые в правиле (функция «ВыполнитьКоманды»).

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

Обработка зависимостей. Функция «ОбработатьЗависимости» проходит по всем зависимостям, перечисленным в правиле, последовательно. Некоторые из них могут быть целями других правил. Для таких зависимостей применяется стандартная процедура выполнения цели (функция «ДостичьЦели»). Зависимости, которые не являются целями, рассматриваются как имена файлов. Для этих файлов проверяется их существование. Если они отсутствуют, процесс завершится с ошибкой в системе make.

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

Обработка команд. На этапе обработки команд решается, следует ли выполнять команды, описанные в правиле. Команды необходимо выполнять в следующих случаях:
- цель является именем действия (абстрактной целью);
- цель - имя файла и сам файл отсутствует;
- 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 команды не будут выполнены. Алгоритм для принятия решения о необходимости выполнения команд можно представить следующим образом:

```
НужноВыполнятьКоманды (Правило) {
    если Правило.Цель.ЯвляетсяАбстрактной () return true
    // цель — имя файла
    если ФайлНеСуществует (Правило.Цель) return true
```

```
            цикл от i=1 до Правило.Число_зависимостей {
            если Правило.Зависимость[ i ].ЯвляетсяАбстрактной () return true
        иначе {
            если ВремяМодификации(Правило.Зависимость[ i ]) > ВремяМодификации (Правило.Цель)
     return true
    }
  }
  return false
}
```

Абстрактные цели и имена файлов. Утилита make различает имена действий и названия файлов следующим образом: сначала происходит поиск файла с указанным именем. Если файл обнаружен, то цель или зависимость трактуются как имя файла. В противном случае это имя считается либо несуществующим файлом, либо именем действия, поскольку различия между этими случаями не предусмотрены и они обрабатываются аналогично.

Стоит упомянуть, что такой метод имеет свои недостатки. Во-первых, make неэффективно использует время, выполняя поиск несуществующих файлов, которые на самом деле представляют собой имена действий. Во-вторых, при таком подходе имена действий не должны совпадать с именами файлов или директорий, иначе выполнение make-файла может привести к ошибкам.

Некоторые версии make предлагают свои решения этой проблемы. Например, в GNU make существует механизм (специальная цель **.PHONY**), который позволяет указать, что конкретное имя является именем действия.

2.1.2. Пример практического применения утилиты make

Пример составления простейшего make-файла

Рассмотрим, как утилита make обрабатывает следующий make-файл на примере кода из листинга 5:

```
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
```

Листинг 5 — Простейший make файл

Предположим, что в каталоге проекта имеются файлы:

main.cpp
Editor.cpp
TextLine.cpp

Если команда make была запущена с помощью `make` без указания цели, то активируется алгоритм выбора цели (функция ВыбратьГлавнуюЦель»), где основной целью становится файл iEdit (первая цель первого правила). Этот файл передается функции «ДостичьЦели», которая находит соответствующее правило в make-файле и инициирует его обработку (функция «ОбработатьПравило»).

Сначала производятся проверки зависимостей, начиная с main.o. Поскольку правило для main.o существует, оно также передается в «ДостичьЦели». Найдено второе правило, где единственной зависимостью является main.cpp. Проверка наличия этого файла на диске подтверждает его существование.

Теперь рассмотрим, что происходит, если команда make запускается с указанием make clean.

В данном случае главной целью становится clean. Процесс обработки проходит аналогично, и все указанные команды исполняются, что приводит к удалению объектных файлов. После этого выполнение make завершается.

Функция «ДостичьЦели» ищет правило, где указана цель main.o, которая находится во втором правиле make-файла. Для этого правила активируется функция «ОбработатьПравило», инициирующая процесс проверки зависимостей (функция «ОбработатьЗависимости»). В этом правиле указана единственная зависимость – main.cpp. В make-файле такой цели нет, поэтому зависимости main.cpp интерпретируются как имя файла. Затем происходит проверка наличия этого файла на диске с помощью функции «ПроверитьНаличиеФайла» – файл существует, и на этом этапе работа с зависимостями завершается.

После проверки зависимостей функция «ОбработатьПравило» решает, следует ли выполнять команды, указанные в правиле (функция «НужноВыполнятьКоманды»). Поскольку цель (файл main.o) не найдена, команды подлежат исполнению. Функция «ВыполнитьКоманды» запускает компилятор GCC, создавая файл main.o, объектный файл.

Цель main.o достигнута, и процесс возвращается к обработке остальных зависимостей первого правила, включая Editor.o и TextLine.o, для которых выполняются аналогичные действия.

Когда все зависимости (main.o, Editor.o и TextLine.o) обработаны, выясняется необходимость выполнения команд. Так как файл iEdit отсутствует, команды выполняются, и создается исполняемый (бинарный) файл iEdit. Таким образом, главная цель достигнута, и работа программы make завершается.

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 случай, когда выполняется команда make clean.

Цель явно указана, и работа начинается с абстрактной цели clean. Эта цель передается функции «ДостичьЦели», которая находит соответствующее правило – пятое правило make-файла. По найденному правилу запускается процедура обработки (функция «ОбработатьПравило»).

Поскольку в правиле нет зависимостей, make переходит к выполнению указанных команд. Цель является действием, следовательно, команды исполняются, и цель clean считается достигнутой. На этом работа программы make завершается.

Применение переменных. Возможность использования переменных в make-файлах - это удобная и широко применяемая функция данной утилиты. В классических версиях make переменные функционируют аналогично макросам в языке C. Для установки значения переменной применяется оператор присваивания.

Например, выражение

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

устанавливает для переменной obj_list значение «main.o Editor.o TextLine.o» (без кавычек). Пробелы между символом «=» и первым словом не учитываются, как и пробелы после последнего слова. Использовать значение переменной можно через конструкцию

$(имя_переменной).

Например, если файл make выглядит так:

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

на выводе получим строку

. .. src/include.

Переменные могут не только содержать строки текста, но и «ссылаться» на другие переменные. К примеру, при обработке make-файла

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

на экране отобразится

-O3 -pipe.

В большинстве случаев применение переменных помогает упростить make-файл и сделать его более понятным. Чтобы облегчить изменения, можно вынести «ключевые» имена и списки в отдельные переменные, поместив их в начало файла:

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

```
	gcc $(obj_list) -o $(program_name)
```
…

Корректировка такого make-файла для сборки другой программы сводится к редактированию всего пары строк в начале.

Применение автоматических переменных

Автоматические переменные представляют собой специальные переменные, которые автоматически получают определенные значения перед выполнением команд, описанных в правилах. Они могут значительно упростить запись правил.

Например, правило

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

с автоматическими переменными можно переписать следующим образом:

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

В этом случае $^ и $@ являются автоматическими переменными. Переменная $^ обозначает "список зависимостей", и при вызове компилятора GCC она будет указывать на строку "main.o Editor.o TextLine.o". Переменная $@ представляет собой "имя цели" и в данном примере будет означать "iEdit". Если бы мы использовали другую автоматическую переменную, такую как $<, она ссылалась бы на первое имя зависимости, то есть в данном случае на файл main.o.

Иногда применение автоматических переменных становится обязательным, например, в шаблонных правилах.

Шаблонные правила.

Шаблонные правила (implicit rules или pattern rules), представляют собой правила, которые можно применять к целой категории файлов. Это отличает их от традиционных правил, которые описывают связи между конкретными файлами. В стандартных реализациях make используется так называемая «суффиксная» запись шаблонных правил. Например, конструкция

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

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

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

В современных версиях make более предпочтительно записывать цели в виде

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

где используется автоматическая переменная $^ для указания компилятору имени файла-зависимости.

Автоматические переменные представляют собой единственный способ определить файлы, к которым в данный момент применяется шаблонное правило, так как они могут использоваться с различными файлами. Применение шаблонных правил значительно упрощает структуру make-файла и делает его более гибким. Рассмотрим упрощенный проектный файл iEdit: main.o Editor.o TextLine.o, который представлен в листинге 6.

```
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 $^
```

Листинг 6 — Упрощённый проектный файл

Все исходные тексты программы обрабатываются одинаково: для них вызывается компилятор GCC. С использованием шаблонных правил листинг 6 можно переписать следующим образом:

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

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

Пример создания более сложного make-файла

Два предыдущих примера упрощают процесс создания проектов с помощью make-файлов. Важно отметить, что автоматизировать перечисление всех объектных файлов, необходимых для программы, также возможно. Однако использование следующего варианта для создания бинарного файла:

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

может оказаться неэффективным, поскольку в данном случае будут учитываться только те объектные файлы, которые существуют на данный момент. Это особенно актуально при наличии сложных заголовочных файлов (*.h), которые определяют зависимости между

частями проекта. Чтобы избежать такой проблемы, рекомендуется воспользоваться более продвинутым методом, который предполагает, что все файлы с исходным кодом должны быть скомпилированы и объединены в итоговую программу.

Данный подход состоит из двух этапов:

1. Получение списка всех файлов с исходным кодом проекта (всех файлов с расширением «cpp»). Для этого используются функции обработки строк, в частности, функция wildcard, которая формирует список файлов по заданному шаблону в определенной директории.

2. Преобразование списка исходников в список объектных файлов (замена расширения «cpp» на расширение «o»). Для этого используется функция patsubst, которая заменяет указанную подстроку.

В листинге 7 приведена изменённую версию make-файла с учётом указанных этапов.

```
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
```

Листинг 7 — Измененная версия make файла с учётом указанных этапов

Список объектных файлов для программы формируется автоматически (цель iEdit). Вначале с помощью функции wildcard создаётся перечень всех файлов с расширением «cpp», которые находятся в папке проекта. Затем с помощью функции patsubst этот полученный список исходников трансформируется в список объектных файлов, изменяя расширение с «cpp» на «o». Теперь make-файл стал более универсальным.

Автоматическое построение зависимостей от заголовочных файлов

После автоматизации перечисления объектных файлов остаётся актуальной задача определения зависимостей между объектными и заголовочными файлами. Например:

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

Ручное перечисление этих зависимостей может оказаться трудоемким. Достаточно открыть исходный текст и записать все заголовочные файлы, подключенные с помощью директивы «#include», но это не всегда достаточно, так как заголовочные файлы могут включать другие заголовки, что создает «цепочку зависимостей», которая может быть довольно длинной. Для формирования списка зависимостей можно использовать утилиту make и компилятор GCC.

GCC предоставляет несколько опций для работы с make:

- Флаг -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.

- Флаг -MM. Аналогичен -M, но список включает лишь те файлы, подключенные через «#include «имя_файла».

- Флаг -MD. Похож на -M, однако зависимости записываются в отдельный файл, название которого формируется из имени исходного файла с заменой расширения на «d».

- Флаг -MMD. Аналогичен -MD, но в список зависимостей входят только определенные файлы.

Как видно, компилятор может выдавать только список зависимостей и заканчивать работу (опции -M и -MM) или продолжать обычную компиляцию, создавая также файл зависимостей (опции -MD и -MMD). Второй вариант предпочтительнее, так как при изменении исходного файла обновляется только соответствующий файл зависимостей.

Построение файлов зависимостей происходит «параллельно» с основной работой компилятора и практически не отражается на времени компиляции.
Для включения файлов зависимостей в утилиту make нужно использовать директиву:

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

Система корректно функционирует лишь при наличии хотя бы одного файла с расширением «d» в каталоге. Если таких файлов не имеется, то команда make завершится с ошибкой, так как она не сможет «собрать» эти файлы. В случае применения символа подстановки, если нужные файлы отсутствуют, данная функция просто выдаст пустую строку. После этого директива include, получившая пустую строку в качестве аргумента, будет проигнорирована и не вызовет никаких ошибок. Таким образом, обновленный вариант makefile из приведенного примера выглядит следующим образом.

Эта конструкция позволит избежать ошибок при отсутствии файлов с расширением «d». Теперь новый makefile может выглядеть так:

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

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

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

При изменении любого из файлов Editor.cpp, Editor.h или TextLine.h, оператор make инициирует перекомпиляцию для получения новой версии файла Editor.o.

Размещение файлов с исходными текстами по директориям

Привиденный выше make-файл полностью функционирует и может быть эффективно использован для компиляции небольших приложений. Тем не менее, по мере роста программы становится неудобно хранить все исходные файлы в одной директории. В таких случаях стоит разместить их в различных каталогах, которые будут соответствовать логической структуре проекта. Для этого нужно модифицировать *make*-файл, чтобы неявное правило

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

осталось рабочим. Для этого применяется переменная VPATH, где указываются директории, содержащие исходные коды. В следующем примере файлы Editor.cpp и Editor.h находятся в папке Editor, а TextLine.cpp и TextLine.h — в каталоге TextLine.

main.cpp
main.h
Editor/
	Editor.cpp
	Editor.h
TextLine /
	TextLine.cpp
	TextLine.h
makefile

Файл make для данного примера приведён в листинге 8.

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

Листинг 8 — Файл make с включением директории.

По сравнению с предыдущей версией make-файла были внесены следующие улучшения:

- введена отдельная переменная source_dirs для хранения списка директорий с исходными файлами, которая используется в нескольких местах;
- шаблон поиска для функции wildcard (переменная search_wildcard s) формируется «динамически» на основе списка source_dirs;
- переменная VPATH используется для нахождения исходных текстов программ;
- компилятору разрешено искать заголовочные файлы во всех директориях, используя функцию addprefix для добавления флага «I» к строке компилятора GCC;
- при формировании списка объектных файлов из имен исходников убирается имя директории с помощью функции notdir;
- для составления списка файлов с расширением «cpp» применяется функция addsuffix, добавляющая суффикс «/*.cpp» к названиям каталогов из переменной source_dirs.

Сборка программы с разными параметрами компиляции (динамическая компиляция)

В некоторых случаях становится необходимым создать несколько версий программы, собранных с использованием разных параметров. Классическим примером служат отладочная и рабочая версии одного и того же приложения. В таких ситуациях следует прибегнуть к подходу, при котором

1) все версии программы компилируются с помощью единого make-файла;

2) нужные настройки компилятора указываются в make-файле, принимая параметры, переданные в командной строке (например, флаги компиляции).

Пример команды: make compile_flags="-O3 -funroll-loops -fomit-frame-pointer". Таким образом, для задания параметров компиляции достаточно добавить переменную compile_flags в makefile. Если флагов несколько (и в строке есть пробелы), значение переменной compile_flags должно быть заключено в кавычки. Пример файла makefile с параметрами представлен в листинге 9.

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

Листинг 9 — Параметризированный make файл

Переменная compile_flags получает значение из командной строки, если оно определено, иначе используется значение по умолчанию — «pipe». Для ускорения компиляции к флагам добавляется ключ pipe. Важно упомянуть, что директива override

необходима для изменения переменной compile_flags внутри make-файла, иначе переданные флаги из командной строки не будут применены.

2.2. **ПОСЛЕДОВАТЕЛЬНОСТЬ ВЫПОЛНЕНИЯ РАБОТЫ**

1. Ознакомиться с теоретическим материалом.

2. Используйте в качестве основы лабораторную работу «**Комбинирование разноязыковых модулей**» для данной работы.

Основное задание:

3. Для автоматизации сборки проекта утилитой make создайте make-файл (см. п. «Пример создания более сложного make-файла»).

4. Выполните программу (скомпилировать, при необходимости выполнить отладку).

5. Покажите, что при изменении одного исходного файла и последующем вызове make будут исполнены только необходимые команды компиляции (неизмененные файлы перекомпилированы не будут) и изменены атрибуты и/или размер объектных файлов (файлы с расширением .o).

Дополнительные задания:

6. Создайте make-файл с высоким уровнем автоматизированной обработки исходных файлов программы согласно следующим условиям:

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

7. Реализуйте возможность **динамического** изменения функционального назначения программы с помощью make, например:

- Управление методом получения данных (выбор между вводом матрицы/массива из консоли, из файла, или генераций с ГПСЧ);
- Управление методом обработки (например, выбор между несколькими способами обработки данных);
- Управление вспомогательным функционалом (логгирование работы или отладочные сообщения);
- Управление кроссплатформленностью. Сделайте возможность запуска программы под различные операционные системы.