make

Практикум, 3 курс

Рассказывает:

Подымов Владислав Васильевич

Осень 2016

Вступление

Зачем нужно знать make?

Чтобы в **Linux**‘е **…**

* не писать одни и те же команды в консоли при сборке бинарников (*программ*)

* упростить сборку больших и распределённых проектов

* автоматизировать исполнение любых консольных команд

* ...

Хотите хорошо работать в Linux’е – учитесь хорошо работать в его консоли

А Windows?

Например, поставить окружение Linux (MinGW, Cygwin, …) и работать в нём

Или изучать самим, что есть подобного в Windows

(*но распределённые проекты чаще пишутся в окружении Linux*)

Что такое make

1. Консольная утилита

```
terminal:> make
echo "Hello, World!"
Hello, World!
terminal:>
```

(*исполняет в консоли запрограммированную последовательность действий*)

2. Файлы, которые читает и исполняет эта утилита

Утилита ищет в текущей папке файл с именем ***makefile***

Утилитой могут восприниматься и другие имена

3. Скриптовый язык, на котором пишутся эти файлы

makefile

```
1 main:
2 	echo "Hello, World!"
3
```

Hello, World!

1. Создаём файл “makefile” с таким текстом

makefile

```
1 main:
2 	echo "Hello, World!"
3
```

Цель

Исполняемые команды (*здесь - одна*)

Табуляция

2. Запускаем консоль

3. Заходим в папку с makefile’ом

4. Набираем команду “make”

5. Получаем вывод как на картинке

```
terminal:> make
echo "Hello, World!"
Hello, World!
terminal:>
```

Какая команда сейчас исполнится

Команда исполняется

Цели

Можно (*и нужно*) делать много целей

(*здесь - две*)

makefile (2)

```
target1:
	echo "First Hello"
target2:
	echo "Second Hello"
```

```
terminal:> make target1
echo "First Hello"
First Hello
terminal:> make target2
echo "Second Hello"
Second Hello
terminal:> make
echo "First Hello"
First Hello
terminal:>
```

При вызове make можно явно указывать,

какая цель должна быть собрана

Самая первая цель в файле – **главная**,

она собирается по умолчанию

Целевой файл

“Хороший стиль” составления makefile’а такой:

- Цель может быть **абстрактной**: не создавать файлов
- Если цель предполагает создание файла, то этот файл **один** (**целевой файл**)
- Можно не соблюдать эти два правила, если понимаете, что делаете (*понятие целевого файла при этом остаётся*)
- Имя целевого файла должно совпадать с названием цели
- При этом команд для создания файла может вызываться много – это нормально

Каждой цели, независимо от того, что она создаёт, сопоставлен целевой файл называющийся так же, как и цель

(*с некоторыми оговорками – об этом будет позже*)

Зависимости

Список зависимостей

```
makefile
1 main: file1 file2
2 »	cat file1
3 »	cat file2
4 file1:
5 »	echo "text of file1" > file1
6 file2:
7 »	echo "text of file2" > file2
8
```

Что такое “зависимости”?

Это **имена файлов**, влияющих на то, что должно быть записано в целевой файл

- Прежде чем собрать цель, make попытается собрать все зависимости (*считая их именами целевых файлов*)
- Если хотя бы одна зависимость не удалось собрать, цель не соберётся

Хотим собрать цель main

Сначала собираем цели file1 и file2

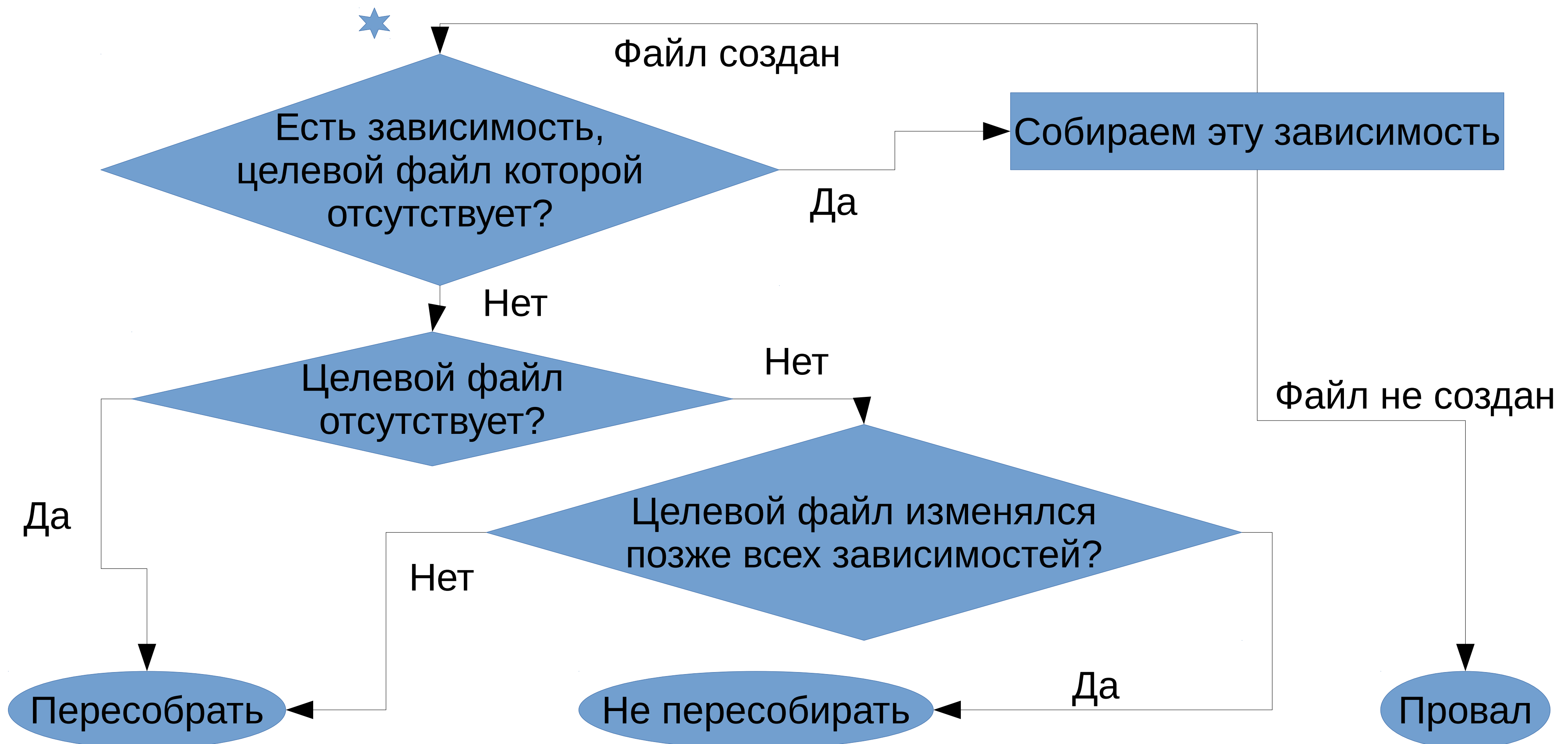
Успех, собираем цель main

```
terminal:> make
echo "text of file1" > file1
echo "text of file2" > file2
cat file1
text of file1
cat file2
text of file2
terminal:>
```

Ленивая сборка

- Абстрактная цель пересобирается всегда
- make **не пересобирает** цели, про которые может точно утверждать, что целевой файл никак не изменится при пересборке

И как же make это делает?



.PHONY

- Это зарезервированное имя цели

  (*не рекомендуется делать цель, собирающую файл .PHONY*)

- В зависимостях этой цели указываются другие абстрактные цели
- Эта цель не будет собираться через зависимости

Так ведь и без .PHONY всё заработает?

Есть как минимум две причины явно указывать абстрактные цели

1. **Эффективность**:

   make может отработать быстрее, распараллеливая явные абстрактные цели

2. **Корректность**:

echo makefile

Плохо

makefile

```
1 echo:
2 »       echo "It's working!"
3
```

```
terminal:> make
make: 'echo' is up to date.
terminal:>
```

Хорошо

makefile

```
1 echo:
2 »       echo "It's working!"
3
4 .PHONY: echo
5
```

```
terminal:> make
echo "It's working!"
It's working!
terminal:>
```

Типичный makefile

```
main: main.o aux.o auxx.o
	g++ -o main main.o aux.o auxx.o

main.o: main.cpp aux.hpp auxx.hpp
	g++ -c main.cpp

aux.o: aux.hpp aux.cpp
	g++ -c aux.cpp

auxx.o: auxx.hpp auxx.cpp
	g++ -c auxx.cpp

clean: cleanobj cleanmain

cleanobj:
	rm *.o

cleanmain:
	rm main

.PHONY: clean cleanobj cleanmain
```

```
terminal:> ls
aux.cpp  aux.hpp  auxx.cpp  auxx.hpp  main.cpp  makefile
terminal:> make
g++ -c main.cpp
g++ -c aux.cpp
g++ -c auxx.cpp
g++ -o main main.o aux.o auxx.o
terminal:> ls
aux.cpp  aux.o     auxx.hpp  main      main.o
aux.hpp  auxx.cpp  auxx.o    main.cpp  makefile
terminal:> make cleanobj
rm *.o
terminal:> ls
aux.cpp  aux.hpp  auxx.cpp  auxx.hpp  main  main.cpp  makefile
terminal:> make cleanmain
rm main
terminal:>
```

Переменные

Эти три makefile’a работают абсолютно одинаково:

```
main: main.o aux.o auxx.o
	gcc -o main main.o aux.o auxx.o

```

```
OBJECTS = main.o aux.o auxx.o

main: $(OBJECTS)
	gcc -o main $(OBJECTS)

```

Переменная

```
OBJECTS = main.o aux.o auxx.o
C = c

main: $(OBJECTS)
	g$(C)$(C) -o main $(OBJECTS)

```

Внешние переменные

Переменные окружения консоли (*кто не знает, спросите у интернета, что это*) можно использовать в качестве переменных makefile'а

Например, такие два вызова make отрабатывают одинаково:

```
main: main.cpp
	gcc -o main main.cpp

```

```
terminal:> make
gcc -o main main.cpp
terminal:>
```

```
$(TARGET): $(SOURCE)
	gcc -o $(TARGET) $(SOURCE)

```

```
terminal:> TARGET=main SOURCE=main.cpp make
gcc -o main main.cpp
terminal:>
```

Автоматические переменные

В языке make есть **много** зарезервированных имён переменных

Среди них есть переменные, значение которых определяется текущей целью, например:

- **$^** - список имён всех зависимостей
- **$<** - имя первой зависимости
- **$@** - имя текущей цели

В частности, такие makefile’ы работают одинаково:

```
main: main.cpp aux.o auxx.o
	g++ -o main main.cpp aux.o auxx.o

aux.o: aux.cpp aux.hpp
	g++ -c aux.cpp

auxx.o: auxx.cpp auxx.hpp
	g++ -c auxx.cpp
```

```
main: main.cpp aux.o auxx.o
	g++ -o $@ $^

aux.o: aux.cpp aux.hpp
	g++ -c $<

auxx.o: auxx.cpp auxx.hpp
	g++ -c $<
```

Обработка переменных

Похожим на переменные способом можно в makefile’е применять **функции**

В том числе в языке make есть **много** зарезервированных имён функций

Например:

$(subst from,to,text) – это слово text, в котором каждое вхождение from заменено на to

$(patsubst pat,repl,text) в списке слов text (*разделитель – пробел*)

заменяет каждое слово из text, удовлетворяющее шаблону pat, на repl

В pat и repl можно использовать специальный символ ‘%’:

в pat это “любой набор символов”,

а в repl - “тот же набор символов, что был в pat”

А ещё символ **%** можно писать в названиях цели и зависимостей,
и это означает “любая цель, попадающая под шаблон” (как в pat и repl)

$(var:pat=repl) – синоним $(patsubst %pat,%repl,$(var))

$(addsuffix suffix,names) – добавить суффикс suffix к каждому слову списка names

Пример применения функций

makefile с функциями ***лучше*** **(?)** makefile’а без функций:

```
main: main.cpp aux.o auxx.o
	g++ -o $@ $^

aux.o: aux.cpp aux.hpp
	g++ -c $@

auxx.o: auxx.cpp auxx.hpp
	g++ -c $@
```

```
objfiles = aux auxx

main: main.cpp $(obj:=.o)
	g++ -o $@ $^

%.o: %.cpp %.hpp
	g++ -c $@
```

Автоматическая сборка

В некоторых случаях make сам знает, какие команды применять для сборки цели

В таких случаях можно опускать список команд (*и надеяться, что всё заработает*)

Например, makefile для сквозного примера с main.cpp, aux*, auxx*
может выглядеть так:

```
1  obj = aux auxx
2
3  main: main.cpp $(obj:=.o) $(obj:=.hpp)
4  %.o: %.cpp %.hpp
5
```

Make-рекурсия

Ещё одно “правило хорошего тона” при написании makefile’ов:

если проект состоит из нескольких логически обособленных папок,
то следует написать свой makefile для каждой папки

и вызывать эти makefile’ы из корневой папки:

```
1 subdirs = dir1 dir2 dir3
2
3 .PHONY: $(subdirs)
4
5 $(subdirs):
6 »       $(MAKE) -C $@
```

Создаются цели для dir1, dir2, dir3
с одним и тем же телом

Вызвать ту же make-команду, что и текущая

Сменить папку на ...

А как лучше соблюсти зависимости, относящиеся к подпапкам?

И как это сочетать с .PHONY? В каких случаях эту цель лучше убрать?

Заключение

А насколько важно соблюдать все “правила хорошего тона”?

<u>Абсолютно неважно</u>

Главное – чтобы

- Всё работало
- Программисты, вынужденные работать с тем, что вы понаписали,
  не покалечили вас, пока вы спите

Практическое задание

Написать **без использования автоматической сборки**

как можно лучше makefile’ы, собирающие исполняемые файлы

prog1, prog2 из соответствующих .cpp-файлов

и стирающие их и файлы .o,
если исходные файлы имеют такую структуру: