make

Многофайловый проект

Компиляция

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
gcc -std=c99 -Wall -Werror -pedantic -c bye.c
gcc -std=c99 -Wall -Werror -pedantic -c main.c
gcc -std=c99 -Wall -Werror -pedantic -c test.c
```

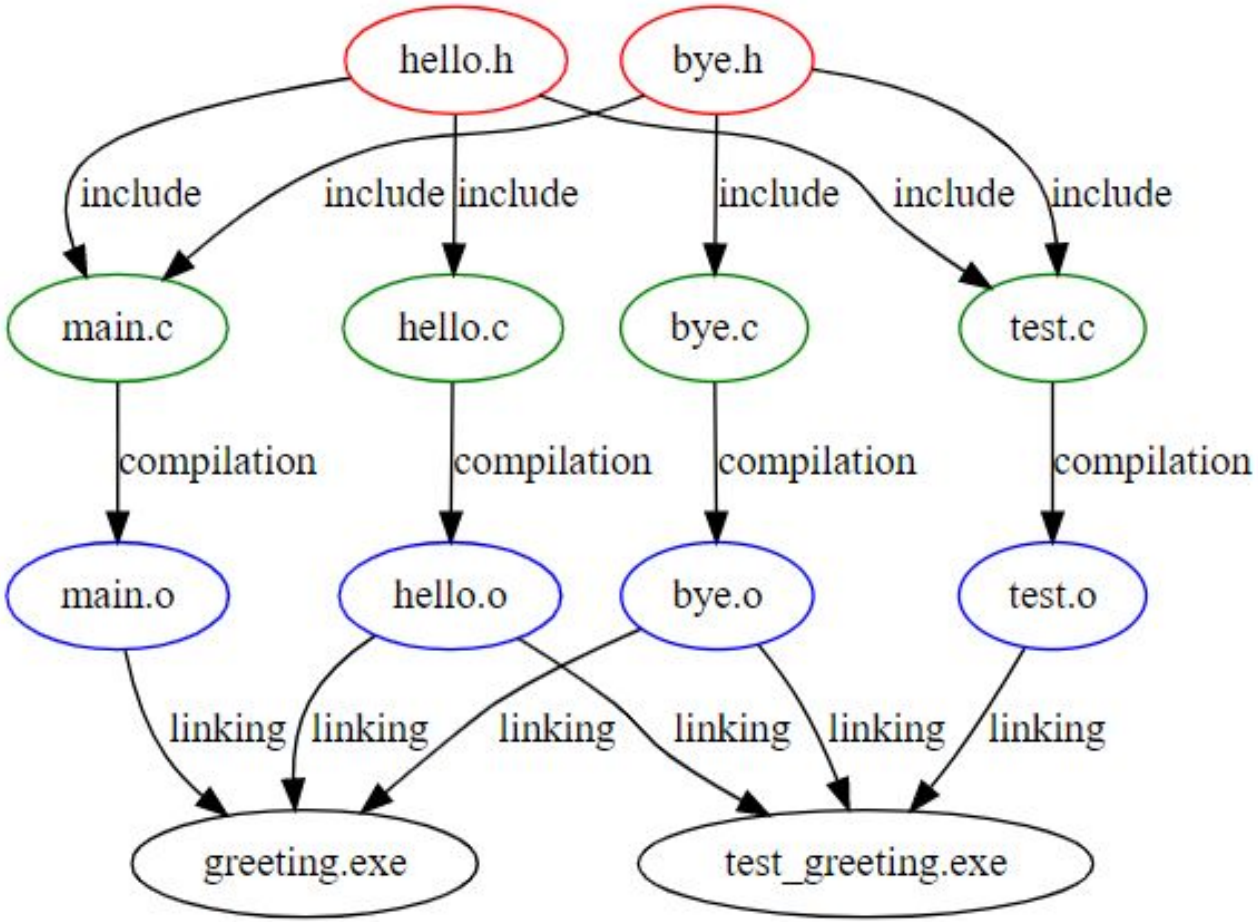
Компоновка

```
gcc -o greeting.exe hello.o bye.o main.o
gcc -o test_greeting.exe hello.o bye.o test.o
```

Почему плохо делать так?

```
gcc -std=c99 -Wall -Werror *.c -o app.exe
```

Граф зависимостей



Утилита make

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую.

- GNU Make (рассматривается далее)
- BSD Make
- Microsoft Make (nmake)

Принципы работы

Необходимо создать так называемый *сценарий сборки проекта* (make-файл). Этот файл описывает

- отношения между файлами программы;
- содержит команды для обновления каждого файла.

Утилита make использует информацию из make-файла и время последнего изменения каждого файла для того, чтобы решить, какие файлы нужно обновить.

Сценарий сборки проекта

цель: зависимость_1 ... зависимость_n

[tab]команда_1

[tab]команда_2

...

[tab]команда_m

что создать/сделать: из чего создать

как создать/что сделать

Простой сценарий сборки

```
greeting.exe : hello.o bye.o main.o
    gcc -o greeting.exe hello.o bye.o main.o

test_greeting.exe : hello.o bye.o test.o
    gcc -o test_greeting.exe hello.o bye.o test.o

hello.o : hello.c hello.h
    gcc -std=c99 -Wall -Werror -pedantic -c hello.c

bye.o : bye.c bye.h
    gcc -std=c99 -Wall -Werror -pedantic -c bye.c

main.o : main.c hello.h bye.h
    gcc -std=c99 -Wall -Werror -pedantic -c main.c

test.o : test.c hello.h bye.h
    gcc -std=c99 -Wall -Werror -pedantic -c test.c

clean :
    rm *.o *.exe
```

Алгоритм работы make (1)

Первый запуск make

- make читает сценарий сборки и начинает выполнять первое правило

```
greeting.exe : hello.o bye.o main.o
    gcc -o greeting.exe hello.o bye.o main.o
```

- Для выполнения этого правила необходимо сначала обработать зависимости

```
hello.o bye.o main.o
```

- make ищет правило для создания файла hello.o

```
hello.o : hello.c hello.h
    gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

Алгоритм работы make (2)

Первый запуск make

- Файл hello.o отсутствует, файлы hello.c и hello.h существуют. Следовательно, правило для создания hello.o может быть выполнено

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

- Аналогично обрабатываются зависимости bye.o и main.o.
- Все зависимости получены, теперь правило для построения greeting.exe может быть выполнено

```
gcc -o greeting.exe hello.o bye.o main.o
```

Алгоритм работы make (3)

Второй запуск make (hello.c был изменен)

- make читает сценарий сборки и начинает выполнять первое правило

```
greeting.exe : hello.o bye.o main.o
    gcc -o greeting.exe hello.o bye.o main.o
```

- Для выполнения этого правила необходимо сначала обработать зависимости

```
hello.o bye.o main.o
```

- make ищет правило для создания файла hello.o

```
hello.o : hello.c hello.h
    gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

Алгоритм работы make (4)

Второй запуск make (hello.c был изменен)

- Файлы hello.o, hello.c и hello.h существуют, но время изменения hello.o меньше времени изменения hello.c. Придется пересоздать файл hello.o

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

- Аналогично обрабатываются зависимости bye.o и main.o, но эти файлы были изменены позже соответствующих си-файлов, т.е. ничего делать не нужно.

Алгоритм работы make (5)

Второй запуск make (hello.c был изменен)

- Все зависимости получены. Время изменения greeting.exe меньше времени изменения hello.o. Придется пересоздать greeting.exe

```
gcc -o greeting.exe hello.o bye.o main.o
```

Ключи запуска make

- Ключ «-f» используется для указания имени файла сценария сборки

```
make -f makefile_2
```

- Ключ «-B» используется для безусловного выполнения правил

```
make -B
```

- Ключ «-n» используется для вывода команд без их выполнения

```
make –n
```

- Ключ «-i» используется для игнорирования ошибок при выполнении команд

```
make -i
```

Использование переменных и комментариев (1)

Строки, которые начинаются с символа '#', являются комментариями.

Определить переменную в make-файле можно следующим образом:

```
VAR_NAME := value
```

Чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '$'.

```
$(VAR_NAME)
```

Использование переменных и комментариев (2)

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
    $(CC) -o greeting.exe $(OBJS) main.o

test_greeting.exe : $(OBJS) test.o
    $(CC) -o test_greeting.exe $(OBJS) test.o

hello.o : hello.c hello.h
    $(CC) $(CFLAGS) -c hello.c
```

Использование переменных и комментариев (3)

```
bye.o : bye.c bye.h
    $(CC) $(CFLAGS) -c bye.c

main.o : main.c hello.h bye.h
    $(CC)  $(CFLAGS) -c main.c

test.o : test.c hello.h bye.h
    $(CC)  $(CFLAGS) -c test.c

clean :
    rm *.o *.exe
```

Фиктивные (.PHONY) цели

В make-файле могут встречаются цели, которые не являются именами файлов. Такие цели называются *фиктивными* и используются для выполнения каких-то действий (очистки, установки и т.п.).

Чтобы make даже не пытался интерпретировать таких как цели как имена файлов их помечают атрибутом .PHONY.

```
.PHONY: clean
```

Неявные правила и переменные

```
# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
    $(CC) -o greeting.exe $(OBJS) main.o

test_greeting.exe : $(OBJS) test.o
    $(CC) -o test_greeting.exe $(OBJS) test.o

.PHONY : clean
clean :
    $(RM) *.o *.exe
```

Ключ «-p» показывает неявные правила и переменные. Ключ «-r» запрещает использовать неявные правила.

Автоматические переменные (1)

Автоматические переменные - это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд.

- Переменная "$^" означает "список зависимостей".
- Переменная "$@" означает "имя цели".
- Переменная "$<" является просто первой зависимостью.
- …

Автоматические переменные (2)

Было

```
greeting.exe : $(OBJS) main.o
	$(CC) -o greeting.exe $(OBJS) main.o
```

Стало

```
greeting.exe : $(OBJS) main.o
	$(CC) -o $@ $^
```

Было

```
hello.o : hello.c hello.h
	$(CC) $(CFLAGS) -c hello.c
```

Стало

```
hello.o : hello.c hello.h
	$(CC) $(CFLAGS) -c $<
```

Автоматические переменные (3)

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
    $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

hello.o : hello.c hello.h
    $(CC) $(CFLAGS) -c $<
```

Автоматические переменные (4)

```
bye.o : bye.c bye.h
    $(CC) $(CFLAGS) -c $<

main.o : main.c hello.h bye.h
    $(CC) $(CFLAGS) -c $<

test.o : test.c hello.h bye.h
    $(CC) $(CFLAGS) -c $<

.PHONY : clean
clean :
    $(RM) *.o *.exe
```

Шаблонные правила (1)

%.расш_файлов_целей : %.расш_файлов_зав

[tab]команда_1

[tab]команда_2

...

[tab]команда_m

Шаблонные правила (2)

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
    $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

%.o : %.c *.h
    $(CC) $(CFLAGS) -c $<

.PHONY : clean
clean :
    $(RM) *.o *.exe
```

Сборка программы с разными параметрами компиляции (1)

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

ifeq ($(mode), debug)
    # Отладочная сборка: добавим генерацию отладочной информации
    CFLAGS += -g3
endif

ifeq ($(mode), release)
```

Сборка программы с разными параметрами компиляции (2)

```
    # Финальная сборка: исключим отладочную информацию и
    # утверждения (asserts)
    CFLAGS += -DNDEBUG -g0
endif

greeting.exe : $(OBJS) main.o
    $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

%.o : %.c *.h
    $(CC) $(CFLAGS) -c $<

.PHONY : clean
clean :
    $(RM) *.o *.exe
```

Присваивание переменных, зависящих от цели (1)

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

debug : CFLAGS += -g3
debug : greeting.exe

release : CFLAGS += -DNDEBUG -g0
release : greeting.exe
```

Присваивание переменных, зависящих от цели (2)

```
greeting.exe : $(OBJS) main.o
    $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

%.o : %.c *.h
    $(CC) $(CFLAGS) -c $<

.PHONY : clean debug release
clean :
    $(RM) *.o *.exe
```

Генерация зависимостей (1)

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

# Все c-файлы (или так SRCS := $(wildcard *.c))
SRCS := hello.c bye.c test.c main.c

greeting.exe : $(OBJS) main.o
    $(CC) $^ -o $@
```

Генерация зависимостей (2)

```
test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

%.o : %.c
    $(CC) $(CFLAGS) -c $<

%.d : %.c
    $(CC) -M $< > $@

# $(SRCS:.c=.d) – заменяет в переменной SRCS имена файлов с
# с расширением "c" на имена с расширением "d"
include $(SRCS:.c=.d)

.PHONY : clean
clean :
    $(RM) *.o *.exe *.d
```

Особенности выполнения команд

- Ненулевой код возврата может прервать выполнение сценария.
- Каждая команда выполняется в своем shell.