# 01. Основные инструменты разработчика в UNIX-подобных ОС

Егор Орлов

Курс: UNIX-DEV-SYS. Системное программирование в среде UNIX (Linux/FreeBSD). ВИШ СПбПУ, 2021

## Содержание

**1 Компилятор** **2**
1.1 Компилятор языка C . . . 2
1.1.1 Первый пример . . . 3
1.2 Стадии компиляции (gcc) . . . 3
1.2.1 Пример 2 . . . 4
1.3 Необходимость в стадиях компиляции . . . 5

**2 Многомодульная сборка проекта** **5**
2.1 Заголовочные файлы . . . 5
2.1.1 Пример 3 . . . 5
2.2 Последовательность сборки (статическая компоновка) . . . 6
2.3 Последовательность сборки (динамическая комоновка) . . . 6
2.4 Runtime-компоновка (libdl) . . . 8
2.4.1 Пример 4 . . . 8

**3 Сборщик make** **9**
3.1 Необходимость . . . 9
3.2 Структура make-файла . . . 9
3.3 Встроенная база правил make . . . 9
3.4 Makefile для примера 3 (динамическая компоновка) . . . 9
3.5 Фиктивные (PHONY) цели . . . 10
3.6 Makefile для примера 4 (runtime-компоновка) . . . 10
3.7 Переменные Makefile . . . 10
3.8 Автоматические переменные . . . 11
3.9 Makefile с автоматическими переменными . . . 11

**4 Система контроля версий (git)** **11**
4.1 Установка . . . 11
4.2 Начальная пользовательская настройка . . . 12
4.3 Создание репозитория . . . 12
4.4 Определение состояния репозитория . . . 12
4.5 Состояния файлов . . . 12
4.6 Добавляем в репозиторий файлы . . . 12

4.7 Фиксация изменений (Commit) . . . 13
4.8 Игнорируемые файлы . . . 13
4.9 Попадание в репозиторий дальнейших изменений . . . 13
4.10 Восстановить старую версию файла . . . 13

**5 Работа с удаленным репозиторием** **14**
5.1 Генерация пользовательских SSH-ключей . . . 14
5.2 Создание ssh-репозитория . . . 14
5.3 Настройка на удаленный репозиторий . . . 14
5.3.1 Пример - собственный ssh-репозиторий . . . 14
5.3.2 Пример - репозиторий на GitHub (https) . . . 14
5.3.3 Пример - репозиторий на GitHub (ssh) . . . 15
5.4 Отправка изменений в удаленный репозиторий . . . 15
5.4.1 Пример . . . 15
5.5 Особенности работы с github . . . 15
5.6 Получение изменений из удаленного репозитория . . . 15
5.7 Клонирование удаленного репозитория . . . 15
5.8 Задание. Клонирование удаленного репозитория и работа с локальной копией . . . 16
5.9 Задание. Создание репозитория на github и работа с ним . . . 16

**6 Ветвление** **16**
6.1 Создание новой ветки . . . 16
6.2 Слияние веток . . . 17

# 1. Компилятор

## 1.1. Компилятор языка C

- **Компилятор** - программа, переводящая текст, написанный на языке программирования, в набор машинных кодов
- **cc** - C Compiler - компилятор языка C
  - **gcc** - GNU C Compiler
  - **clang** - транслятор C/C++/Objective C в байт-код LLVM
  - множество коммерческих решений (Intel, Microsoft, etc)

```
$ file $(which cc)
/usr/bin/cc: symbolic link to `gcc'
```

Часто для вызова компилятора используется синоним **cc**

```
$ gcc --version
x86_64-alt-linux-gcc (GCC) 10.3.1 20210703 (ALT Sisyphus 10.3.1-alt2)
```

Версия компилятора gcc, установленная в системе

```
$ cc simple.c
$ clang simple.c
```

Компиляция программы на C

- **a.out** - имя скомпилированного файла по-умолчанию

```
$ cc -o simple simple.c
```

**-o** - указание имени результирующего файла

```
$ cc -v -o simple simple.c
```

**-v** - подробно что делает

### 1.1.1. Первый пример

```
#include <stdio.h>

int main()
{
        printf("1st sample\n");
        return 0;
}
```

## 1.2. Стадии компиляции (gcc)

1. **Препроцессирование** - обработка расширения языка (директив препроцессора)

```
$ gcc -E simple.c
```

Остановиться после препроцессирования

2. **Компиляция** - преобразование в ассемблерный код
   - **file.c -> file.s**

```
$ gcc -S simple.c
```

Остановиться после компиляции

3. **Ассемблирование** - получение объектного файла программы
   - **file.s -> file.o**

```
$ cc -c simple.c
$ file simple.o
simple.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

Остановиться после ассемблирования. Запуск компилятора без этапа компоновки

- Просмотр заголовков объектного файла (objdump)

```
$ objdump -x simple.o
. . .
SYMBOL TABLE:
. . .
0000000000000000 g     F .text  0000000000000017 main
. . .
```

Функция **main** присутствует в сегменте кода - **.text**

- Просмотр содержимого таблицы символов (nm - name management)

```
$ nm simple.o
                 U _GLOBAL_OFFSET_TABLE_
```

```
0000000000000000 T main
                 U puts
```

Функция **main** - в сегменте текста, функция **puts** - нужна, но недоступна, т.е. она должна быть в другом объектном файле и будет добавлена на этапе компоновки.

4. **Компановка(линковка/сборка)** - объединение объектных файлов
   - объектные файлы программы
   - **libc**
   - **crt*** - с runtime/исполняющая система - адаптация кода под работу в ОС
   - ...

```
$ cc -o simple simple.c
$ nm simple
. . .
```

Соглашение об именовании в языке **Си** такое, что компилятор оставляет имена функций такими же, как это указано в коде программе

В языке **C++** используется mangling, т.е. компилятор кодирует в именах функций информацию о типе

### 1.2.1. Пример 2

- Выносим печать в отдельный файл (simple2.c)

```
int main()
{
        print_message("2st sample");
        return 0;
}
```

- Реализация функции печати (print.c)

```
#include <stdio.h>

void print_message(const char *name)
{
  printf("%s\n", name);
}
```

- Пробуем собрать

```
$ cc -c print.c
$ nm print.o
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T print_message
                 U puts
$ cc -c simple2.c
simple2.c: In function ‘main’:
simple2.c:3:10: warning: implicit declaration of function ‘print_message’ [-Wimplicit-function-declarati
    3 |          print_message("2st sample");
      |          ^~~~~~~~~~~~~
$ nm simple2.o
```

```
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U print_message
$ cc simple2.c
simple2.c: In function ‘main’:
simple2.c:3:10: warning: implicit declaration of function ‘print_message’ [-Wimplicit-function-declarati
    3 |          print_message("2st sample");
      |          ^~~~~~~~~~~~~
ld: /tmp/.private/egor/ccYenvtW.o: в функции «main»:
simple2.c:(.text+0x11): неопределённая ссылка на «print_message»
collect2: error: ld returned 1 exit status
```

Видим предупреждение компилятора и ошибку компоновщика

- При сборке многомодульного проекта необходимо сообщать компоновщику расположение всех модулей

```
$ cc print.c simple2.c -o simple2
simple2.c: In function ‘main’:
simple2.c:3:10: warning: implicit declaration of function ‘print_message’ [-Wimplicit-
function-declaration]
    3 |          print_message("2st sample");
      |          ^~~~~~~~~~~~~
```

Видим предупреждение, но сборка состоялась

### 1.3. Необходимость в стадиях компиляции

- Первые 3 стадии компиляции - наиболее времязатратная часть всей процедуры, по сравнению с 4ой
- В больших, многомодульных проектах глупо пересобирать все модули при любом изменении
- Удобно использовать результаты предыдущих сборок (объектные файлы модулей), если исходный код этих модулей не поменялся

```
$ cc -o simple2 simple2.o print.o
```

## 2. Многомодульная сборка проекта

### 2.1. Заголовочные файлы

- Содержат интерфейсную часть модулей, т.е. определения функций, так же это могут быть определения типов, глобальные переменные и т.п.
- Причина: необходимость отдельного описания интерфейсов, чтобы объединение многомодульного проекта на 4ой стадии происходило без сюрпризов.

#### 2.1.1. Пример 3

- Заголовочный файл **print.h**

```
#ifndef __HELLO__
#define __HELLO__
```

```
void print_message(const char *name);

#endif
```

Для линковки модулей, содержащих код на C++ в программу на C перед объявлениями функций пишется

```
extern "C" void print_message(const char *name);
```

Т.к. кодирование типа компилятором C++

- Файл модуля **print.c**

```
#include "print.h"
#include <stdio.h>
void print_message(const char *name)
{
        printf("%s\n", name);
}
```

- Файл основной программы **simple3.c**

```
#include "print.h"
int main()
{
         print_message("3st sample");
         return 0;
}
```

## 2.2. Последовательность сборки (статическая компоновка)

- По шагам

```
$ cc -c print.c
$ cc -c simple3.c
$ cc -o simple3 print.o simple3.o
```

- Или сразу

```
$ cc -o simple3 print.c simple3.c
```

## 2.3. Последовательность сборки (динамическая комоновка)

- Создаем динамическую библиотеку, принято использовать префикс **lib**

```
$ cc -o libPrint.so -shared -fPIC print.c
$ nm libPrint.so
. . .
0000000000001105 T print_message
. . .
```

**-shared** - создать динамическую библиотеку

**-fPIC** - **Position Independent Code** - необходимый режим создания объектного файла для возможности его одновременного использования несколькими процессами, что актуально как раз таки для динамических библиотек

- Собираем основную программу с использованием динамической библиотеки

```
$ cc simple3.c -L. -lPrint -o simple3
$ nm simple3
. . .
0000000000001135 T main
                 U print_message
. . .
```

**-L** - где искать динамические библиотеки

**-l** - какие динамические библиотеки слинковать (без lib и без .so)

**-o** д.б. в конце, а исходник в начале (в данном случае последовательность важна). **И вообще лучше взять за правило - исходник в начале, результат в конце.**

- Запускаем

```
$ ./simple3
./simple3: error while loading shared libraries: libPrint.so: cannot open shared object file: No such fi
```

Системный динамический линковщик при запуске не может найти динамическую библиотеку. Т.к. она находится не по стандартному пути

- Какие динамические библиотеки хочет использовать программа

```
$ ldd simple3
        linux-vdso.so.1 (0x00007ffde6df2000)
        libPrint.so => not found
        libc.so.6 => /lib64/libc.so.6 (0x00007fe2f8cbe000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe2f8eb9000)
```

**ld-linux** - компоновщик Linux **libc** - стандартная библиотека C **linux-vdso** - шлюз в ядро ОС

- Объяснить динамическому линковщику, где нужная библиотека
  - положить ее в стандартное местоположение
  - прописать в настройках динамического линковщика **/etc/ld.so.conf** и заставить динамический линковщик перечитать конфигурацию средствами **ldconfig**
  - указать через переменную окружения

`LD_LIBRARY_PATH`

Позволяет задать дополнительные пути для поиска

`LD_PRELOAD`

Позволяет задать пути по которым библиотеки ищутся перед тем как идет обращение по стандартным путям

- Объявляем значение переменной и запускаем

```
$ export LD_LIBRARY_PATH=.
$ ldd simple3
        linux-vdso.so.1 (0x00007fffb77e3000)
        libPrint.so => ./libPrint.so (0x00007f38f8ea0000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f38f8cac000)
```

```
        /lib64/ld-linux-x86-64.so.2 (0x00007f38f8eac000)
$ ./simple3
```

Все ОК. Библиотека найдена, и более того - уже в памяти (есть адрес). Приложение запускается.

## 2.4. Runtime-компоновка (libdl)

- Подключение библиотек в режиме **runtime**
  - например для реализации системы плагинов для приложения

### 2.4.1. Пример 4

- Код основной программы (simple4.c)

```
#include <stddef.h>
#include <stdbool.h>
#include <stdio.h>
#include <dlfcn.h>

void (*print_message)(const char *);

bool load_library() {
        void *hdl = dlopen("./libPrint.so", RTLD_LAZY);
        if ( hdl == NULL )
                return false;
        print_message = (void (*)(const char*))dlsym(hdl,"print_message");
        if ( print_message == NULL )
                return false;
        return true;
}

int main()
{
        if ( load_library() )
                print_message("4st sample");
        else
                printf("Cannot find library\n");
        return 0;
}
```

- Для сборки требуется подключение библиотеки **libdl**

```
$ cc simple4.c -ldl -o simple4
$ ./simple4
$ ldd simple4
        linux-vdso.so.1 (0x00007ffd0f393000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f0b7e4cd000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f0b7e305000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f0b7e506000)
```

Приложение запускается и использует библиотеку, но от юиюлиотеки не зависит

# 3. Сборщик make

## 3.1. Необходимость

- Простое управление процессом сборки и пересборки ПО с учетом зависимостей между программными модулями

  Никто не пишет команды сборки руками, т.к. это утомительно

## 3.2. Структура make-файла

- Называется Makefile

```
цель1: зависимости
<TAB> команды сборки
<TAB> команды сборки
. . .

цельN: зависимости
<TAB>: команды сборки
. . .
```

## 3.3. Встроенная база правил make

```
$ make -p
```

Наличие встроенных правил позволяет производить простейшие операции сборки однокомпонентных приложений. Либо отдельных модулей многокомпонентного.

```
$ make print.o
cc    -c -o print.o print.c
$ make simple3.o
cc    -c -o simple3.o simple3.c
$ make simple3
cc   simple3.o   -o simple3
ld: simple3.o: в функции «main»:
simple3.c:(.text+0xc): неопределённая ссылка на «print_message»
collect2: error: ld returned 1 exit status
make: *** [<встроенное>: simple3] Ошибка 1
```

Искуственный интеллект не безграничен.

## 3.4. Makefile для примера 3 (динамическая компоновка)

```
all: simple3

simple3: simple3.c print.h libPrint.so
        cc simple3.c -L. -lPrint -o simple3

libPrint.so: print.c print.h
        cc print.c -shared -fPIC -o libPrint.so
```

```
clean:
        -rm -f simple3 libPrint.so *.o 2>/dev/null
```

Первая цель в Makefile является целью по-умолчанию

## 3.5. Фиктивные (PHONY) цели

- Как фиктивные помечаются те цели, которые make не должен искать в процессе обработки Makefile-а. Т.е. цели, не являющиеся файлами

```
.PHONY: all clean install uninstall

all: simple3

simple3: simple3.c print.h libPrint.so
        cc simple3.c -L. -lPrint -o simple3

libPrint.so: print.c print.h
        cc print.c -shared -fPIC -o libPrint.so

clean:
       -rm -f simple3 libPrint.so *.o 2>/dev/null

install:
    install ./simple3 /usr/local/bin
    install libPrint.so /usr/local/lib

uninstall:
    rm -f /usr/local/bin/simple3
    rm -f /usr/local/lib/libPrint.so
```

## 3.6. Makefile для примера 4 (runtime-компоновка)

- **Внимание!** Выполните самостоятельно

## 3.7. Переменные Makefile

- Переменные в Makefile принято указывать в верхнем регистре

```
VAR = value
```

- Получение значения переменной

```
$VAR
```

- Изменяем файл с использованием переменных

```
CFLAGS=-Wall
TARGET=simple3
PREFIX=/usr/local

.PHONY: all clean install uninstall
```

```
all: $(TARGET)

simple3: simple3.c print.h libPrint.so
        cc simple3.c -L. -lPrint -o $(TARGET)

libPrint.so: print.c print.h
        cc print.c -shared -fPIC -o libPrint.so

clean:
       -rm -f $(TARGET) libPrint.so *.o 2>/dev/null

install:
        install $(TARGET) $(PREFIX)/bin
        install libPrint.so $(PREFIX)/lib

uninstall:
        rm -f $(PREFIX)/bin/$(TARGET)
        rm -f $(PREFIX)/lib/libPrint.so
```

- Указание параметров компиляции

`CFLAGS=-Wall`

Включение вывода всех предупреждений - хорошая практика

### 3.8. Автоматические переменные

`$@`

Имя цели обрабатываемого правила

`$<`

Имя первой зависимости обрабатываемого правила

`$^`

Список всех зависимостей обрабатываемого правила

- Использование автоматических переменных упрощает написание Makefile-ов но усложняет их восприятие

### 3.9. Makefile с автоматическими переменными

- **Внимание!** Перепешите Makefile с использованием автоматических переменных самостоятельно.

## 4. Система контроля версий (git)

### 4.1. Установка

- Установка Debian/Ubuntu

```
$ apt-get install git
```

## 4.2. Начальная пользовательская настройка

- Команды настройки

```
$ git config --global user.name "My Name"
$ git config --global user.email email@domain.ru
```

- Пользовательская конфигурация сохраняется в

```
$ cat .gitconfig
[user]
name = My Name
email = email@domain.ru
```

## 4.3. Создание репозитория

- Создание пустого репорзитория в текущем каталоге

```
$ cd project
$ git init
Initialized empty Git repository in /home/user/project/.git/
```

## 4.4. Определение состояния репозитория

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Makefile
    project.c
nothing added to commit but untracked files present (use "git add" to track)
```

- Текущая ветка(branch) репозитория - **master**
- В репозиторий ничего не добавлено (нет коммитов)
- Есть **неотслеживаемые** файлы (все, что были в текущем каталоге)
- Коммитить (добавлять в репозиторий) нечего

## 4.5. Состояния файлов

- **tracked/untracked** - отслеживается или нет эти файлы git-ом, отслеживаются файлы, которые уже есть в репозитории
- **staged/not staged** - файлы, помечены как кандидаты на добавление в репозиторий

## 4.6. Добавляем в репозиторий файлы

- Добавление файлов в список кандидатов на попадение в содержимое репозитория (changes to be commited)

```
$ git add project.c
```

- Добавление всех файлов из каталога

```
$ git add -A
```

- Просмотр отслеживаемых файлов

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   project.c
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Makefile
```

## 4.7. Фиксация изменений (Commit)

- Записываем состояние файлов-кандидатов в репозиторий - фиксируем их изменения

```
$ git commit -m "Initial Commit"
```

## 4.8. Игнорируемые файлы

- Чтобы файлы не светились как **untracked**

```
$ cat .gitignore
project
a.out
*~
.*
*.o
```

## 4.9. Попадание в репозиторий дальнейших изменений

- Все файлы, которые уже есть в репозитории - они в состоянии **tracked**
- Но автоматом они в комиты попадать не будут
- Чтобы файл попал в следующий комит, мы его либо явно добавляем

```
$ git add file.c
```

- Либо создаем коммит указывая необходимость добавить все **tracked**-файлы

```
$ git commit -a -m "Next commit"
```

## 4.10. Восстановить старую версию файла

- При помощи **git log** узнаем идентификатор нужного нам **commit**-а

```
$ git log
```

- При помощи **git checkout** достаем из этого **commit**-а нужный нам файл

```
$ git checkout <id> <file>
```

# 5. Работа с удаленным репозиторием

## 5.1. Генерация пользовательских SSH-ключей

- Требуется для взаимодействия с удаленными репозиториями по протоколу SSH

```
$ ssh-keygen
```

- Публичный и приватный ключ

```
$ ls ~/.ssh
id_rsa id_rsa.pub
```

- Публичный ключ необходимо добавить на удаленный репозиторий, например в профиль на GitHub-е

## 5.2. Создание ssh-репозитория

- Можно создать свой репозиторий на сервере с доступом по SSH

```
$ mkdir project && cd project
$ git init --bare
```

## 5.3. Настройка на удаленный репозиторий

- Добавляем в наш репозиторий связь с удаленным

```
$ git remote add <shortname> <address>
```

- **shortname** - имя удаленного репозитория
- **address** - сетевой путь

### 5.3.1. Пример - собственный ssh-репозиторий

```
$ git remote add project-srv ssh://srv.domain.ru/srv/git/project
$ git remote -v
project-srv       ssh://srv.domain.ru/srv/git/project (fetch)
project-srv       ssh://srv.domain.ru/srv/git/project (push)
```

### 5.3.2. Пример - репозиторий на GitHub (https)

- Репозиторий в аккаунте на github нужно предварительно создать
- Добавление внешнего репозитория

```
$ git remote add origin https://github.com/hse-labs/unix-dev-sys.git
$ git remote -v
origin  https://github.com/hse-labs/unix-dev-sys.git (fetch)
origin  https://github.com/hse-labs/unix-dev-sys.git (push)
```

> В настоящий момент GiHub перестал поддерживать пользовательскую аутентификацию при доступе по HTTPS. Поэтому для загрузки своих изменений в удаленныйй репозиторий проще всего использовать доступ по SSH.

#### 5.3.3. Пример - репозиторий на GitHub (ssh)

```
$ git remote add origin git@github.com:hse-labs/unix-dev-sys.git
$ git remote -v
origin  git@github.com:hse-labs/unix-dev-sys.git (fetch)
origin  git@github.com:hse-labs/unix-dev-sys.git (push)
```

### 5.4. Отправка изменений в удаленный репозиторий

```
$ git push <shortname> <branch>
```

- **shortname** - имя удаленного репозитория (назначенное ему локально)
- **branch** - ветка удаленного репозитория (по-умолчанию - master)

#### 5.4.1. Пример

- Ветка **master** локального репозитория записывается в ветку **master** удаленного

```
$ git push origin master
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1008 bytes | 1008.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:      https://github.com/<login>/<project>/pull/new/master
remote:
To https://github.com/<login>/<project>.git
 * [new branch]      master -> master
```

### 5.5. Особенности работы с github

- Ветка по умолчанию на *github* теперь не **master** а **main**, ибо *инклюзивная терминология*.

```
$ git push origin main
```

- Аутентификация по паролю - *deprecated*. Необходимо настраивать для работы с github-репозиториями более защищенные виды аутентификации

### 5.6. Получение изменений из удаленного репозитория

```
$ git pull project-srv master
```

### 5.7. Клонирование удаленного репозитория

```
$ git clone [-b <branch>] <repository> [<dir>]
```

- Новый локальный репозиторий будет создан автоматически с настройкой на удаленный репозиторий. Из удаленного репозитория будет получена ветка по-умолчанию

```
$ git clone https://github.com/<login>/<project>.git
```

- Если хотим клонирование в текущий каталог (без создания подкаталога)

```
$ git clone https://github.com/<login>/<project>.git ./
```

- Хотим забрать определенную ветку

```
$ git clone -b <branch> https://github.com/<login>/<project>.git ./
```

- Клонирование удаленного репозитория GitHub по SSH

```
$ git clone git@github.com:hse-labs/unix-dev-sys.git
```

### 5.8. Задание. Клонирование удаленного репозитория и работа с локальной копией

1. Выполните клонирование репозитория **hse-labs/unix-dev-sys** по HTTP или SSH к себе на компьютер
2. Добавьте к рабочей копии данных ранее редактируемые файлы (файл Makefile с вашими изменениями)
3. Убедитесь, что добавленные/измененные файлы попадут в комит
4. Выполните комит(фиксацию изменений) в локальном репозитории.

### 5.9. Задание. Создание репозитория на github и работа с ним

1. Выполните генерацию пользовательских SSH-ключей
2. Создайте профиль (зарегистрируйтесь) на GitHub
3. Добавьте свой публичный SSH-ключ в профиль на GitHub
4. Создайте пустой репозиторий для работы в рамках данного курса
5. Настройте свой локальный репозиторий на ваш внешний репозиторий GitHub-а
6. Отправьте ваши локальные данные в созданный внешний репозиторий.

## 6. Ветвление

### 6.1. Создание новой ветки

- Создаем новую ветку в дополнении к стандартной **master** (или **main**)

```
$ git branch task1
```

- Список доступных веток

```
$ git branch
task1
* master
```

- Переключение между ветками (извлечение в рабочий каталог ветки task1)

```
$ git checkout task1
Switched to branch 'task1'
$ git status
On branch task1
nothing to commit, working tree clean
```

## 6.2. Слияние веток

- Работаем в созданной ветке, отслеживаем и подтверждаем изменения, тестируем их успешность, т.е. работа с изменениями закончена, их можно принимать в основную версию проекта

```
$ git add project.c
$ git commit -m "New features"
```

- Переход на основную ветку (основная версия проекта)

```
$ git checkout master
```

- Применение изменений созданной ветки к основной версии проекта

```
$ git merge task1
```

- Далее ветку можно удалить (если не нужна)

```
$ git branch -d task1
```