Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе**

**Дисциплина**: Проектирование ОС и компонентов

**Тема**: Создание инсталлируемой версии утилиты в ОС Windows и Linux

Выполнил студент гр. 13541/4 Степанов Д.С.

(подпись)

Руководитель Душутина Е.В.

(подпись)

“ ” 2017 г.

Санкт - Петербург

2017

Ранее для ОС Windows и Linux было реализовано приложение, осуществляющее отладку программ на языке С++. Необходимо:

* Создать инсталлируемую версию утилиты для ОС Windows
* Создать инсталлируемую версию утилиты для ОС Linux
* Создать версию утилиты в виде службы Windows
* Создать версию утилиты в виде демона Linux

**Выполнение работы**

**Создание инсталлируемой версии в ОС Windows**

В качестве инструмента для создания инсталлируемого файла был выбран NSIS — бесплатный и широко известный инструментарий для создания инсталляционных пакетов в среде Windows. Отличительные особенности — компактность, масштабируемость, поддержка плагинов и локализаций. Из других возможностей NSIS: поддержка сценариев, различных вариантов установки (включая несколько проектов внутри одного инсталлятора), создание веб-инсталляторов.

Дистрибутивы NSIS создаются на основе сценариев. Переменные, функции полностью контролируют как инсталляцию, так и деинсталляцию продукта. С помощью скриптов можно добавлять файлы и директории, вносить изменения в реестр, редактировать текстовые и двоичные файлы, создавать патчи, управлять переменными среды и даже обращаться к Windows API (посредством расширений).

Для создания инсталляторов необходимо задавать константы. Константа в NSIS задается командой !define. Пример:

Определим следующие константы:

;имя приложения

!define PRODUCT\_NAME "Debugger"

;версия приложения

!define PRODUCT\_VERSION "1.0"

;папка, хранящая исходные файлы, подлежащие сжатию.

!define pkgdir "e:\lab3\_files"

Поподробней про последнюю константу. Она служит для указания корневой папки, в которой будут лежать файлы проекта, подлежащие включению в дистрибутив. Это полезно для тех случаев, когда эта папка у вас участвует в нескольких участка кода скрипта, и в случае, если вы переместили исходный набор в другую папку, придется поправить путь только в одном месте.

После того, как мы определили нужные нам константы, подключаем библиотеку «современного» интерфейса пользователя:

!include "MUI.nsh"

Следующее, что нам необходимо сделать, это задать степень сжатия исходных файлов. Делается это командой SetCompressor. NSIS поддерживает несколько методов: ZLIB, BZIP2 и LZMA. Не вдаваясь в технические подробности, скажем, что наиболее оптимальным методом сжатия является именно LZMA, обеспечивающий быструю скорость распаковки и высокий уровень сжатия. Вы можете указать команде опцию /SOLID, указывающую компилятору поместить все сжимаемые данные в один блок, тем самым дополнительно увеличив степень сжатия фалов инсталлятора.

SetCompressor /SOLID lzma

Так как «современный» интерфейс мы подключили, самое время заняться его настройкой. Настраивается он путем задания значений констант уже знакомой нам командой !define.

Константа MUI\_ABORTWARNING определяет, выдавать ли предупреждение при закрытии инсталятора пользователем.

!define MUI\_ABORTWARNING

Константа MUI\_ICON определяет значок инсталятора:

!define MUI\_ICON "${NSISDIR}\Contrib\Graphics\Icons\modern-install.ico"

Здесь ${NSISDIR} - системная константа, означает путь к папке, куда установлен сам NSIS.

Далее необходимо определиться со страницами, которые будут видны конечному пользователю. Они определяются через вызов макросов - логически завершенных блоков кода, определенных, как несложно догадаться, в файле MUI.nsh.

В данном примере нам потребуются:

* Страница приветствия
* Страница прогресса выполнения инсталлятора
* Страница завершения работы

Соответственно, необходимо вставить следующие строки в наш скрипт:

!insertmacro MUI\_PAGE\_WELCOME

!insertmacro MUI\_PAGE\_INSTFILES

!insertmacro MUI\_PAGE\_FINISH

Если мы хотим, чтобы все страницы нашего скрипта были на русском языке, необходимо также добавить вызов макроса локализации:

!insertmacro MUI\_LANGUAGE "Russian"

Команда Name задает название кнопки инсталлятора на панели задач. В нашей случае мы будем использовать её в следующем виде:

Name "${PRODUCT\_NAME}"

Команда Caption задает заголовок окна инсталятора. Её мы зададим равной

Caption "Установка ${PRODUCT\_NAME} ${PRODUCT\_VERSION}"

Имя файла инсталлятора задается командой OutFile. Вот так:

OutFile "${PRODUCT\_NAME}-${PRODUCT\_VERSION}.exe"

На этом в нашем конкретном примере мы закончим определять настройки нашего инсталлятора, и перейдем к самой важной логической составляющей любого инсталлятора на языке NSIS – секции.

Секции — это логический блок команд, наличие которого является обязательным. Инсталлятор должен содержать хотя бы одну секцию. Именно в секциях производятся действия над файлами, операции с реестром, ярлыками и файловой системой.

Блок команд, входящих в секцию, обрамляется ключевыми словами Section- SectionEnd. Дополняем наш скрипт следующими строчками:

Section

SetOutPath "$TEMP"

Команда SetOutPath указывает компилятору, куда производить распаковку файлов. В нашем варианте мы для распаковки файлов используем временную папку системы.

А вот какие файлы подлежат распаковке, определяет команда File:

File "${pkgdir}\Debugger.exe"

Закрываем секцию командой SectionEnd. Полный скрипт:

;имя приложения

!define PRODUCT\_NAME "Debugger"

;версия приложения

!define PRODUCT\_VERSION "1.0"

!define pkgdir "e:\lab3\_files"

!include "MUI.nsh"

SetCompressor /SOLID lzma

!define MUI\_ABORTWARNING

!define MUI\_ICON "${NSISDIR}\Contrib\Graphics\Icons\modern-install.ico"

!insertmacro MUI\_PAGE\_WELCOME

!insertmacro MUI\_PAGE\_INSTFILES

!insertmacro MUI\_PAGE\_FINISH

!insertmacro MUI\_LANGUAGE "Russian"

Name "${PRODUCT\_NAME}"

Caption "Установка ${PRODUCT\_NAME} ${PRODUCT\_VERSION}"

OutFile "${PRODUCT\_NAME}-${PRODUCT\_VERSION}.exe"

Section

SetOutPath "${pkgdir}"

File "${pkgdir}\Debugger.exe"

SectionEnd

На рисунке 1 представлено окно программы NSIS. На рисунках 2-3 представлен получившийся инсталлятор.

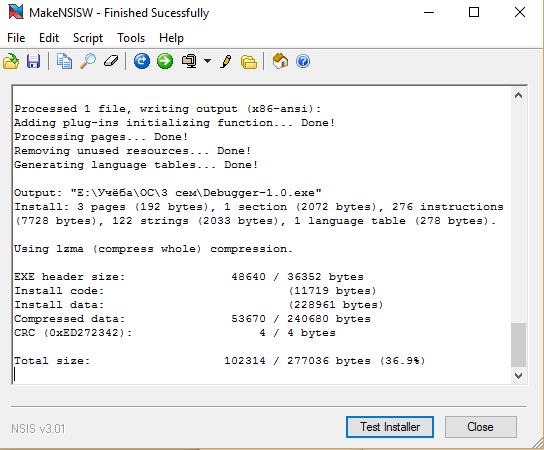


Рис.1. NSIS

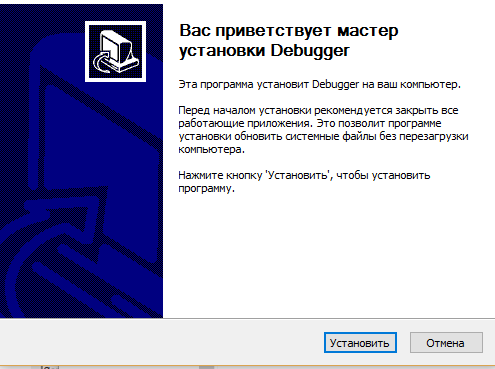


Рис.2. Начальное окно инсталлятора

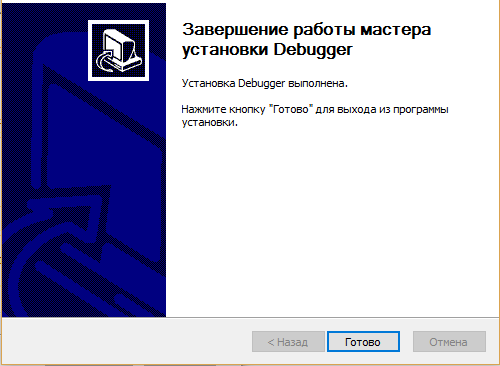


Рис.3. Завершающее окно инсталлятора

**Создание службы в ОС Windows**

Службы операционной системы — это программный процесс, предназначенный для организации взаимодействия различных программных и аппаратных компонентов компьютерной системы. Запуск служб Windows происходит автоматически в момент загрузки компьютера, важнейшие службы продолжают работу до момента его выключения. Остальные службы (активируются) задействуются по мере необходимости.

Каждая служба Windows предназначена для выполнения конкретной операции. Существуют, например, службы обновления, служба времени, служба печати, служба безопасности, сетевые службы Windows и другие.

Автоматический запуск и постоянная работа служб Windows позволяет операционной системе выполнять необходимые операции, причем пользователь не должен явно давать какую-либо команду или запускать соответствующую программу.

Преимущество концепции служб, впервые введенной в операционной системе Windows NT, состоит в том, что становятся доступным независимые средства управления компьютерной системы. Имеются возможность запускать службы, а также корректно останавливать или приостанавливать их. Если на операционную систему возлагаются новые функции, для их выполнения задействуются дополнительные службы.

Работа служб Windows организована по принципу “запрос-ответ“. Приложение обращается к службе для выполнения конкретных действий. Служба активизируется, выполняет запрошенную операцию, сообщает приложению о результате и снова возвращается в пассивное состояние до очередного запроса.

Инсталлятор в виде службы был сделан с помощью плагина для NSIS. Этот плагин содержит основные сервисные функции start, stop и т.д. Полный список функций приведен ниже:

SimpleSC::InstallService [name\_of\_service] [display\_name] [service\_type] [start\_type]

[binary\_path] [dependencies] [account] [password]

SimpleSC::RemoveService [name\_of\_service]

SimpleSC::StartService [name\_of\_service] [arguments] [timeout]

SimpleSC::StopService [name\_of\_service] [wait\_for\_file\_release] [timeout]

SimpleSC::PauseService [name\_of\_service] [timeout]

SimpleSC::ContinueService [name\_of\_service] [timeout]

SimpleSC::RestartService [name\_of\_service] [arguments] [timeout]

SimpleSC::ExistsService [name\_of\_service]

SimpleSC::GetServiceDisplayName [name\_of\_service]

SimpleSC::GetServiceName [display\_name]

SimpleSC::GetServiceStatus [name\_of\_service]

SimpleSC::GetServiceDescription [name\_of\_service]

SimpleSC::GetServiceStartType [name\_of\_service]

SimpleSC::GetServiceBinaryPath [name\_of\_service]

SimpleSC::GetServiceLogon [name\_of\_service]

SimpleSC::GetServiceFailure [name\_of\_service]

SimpleSC::GetServiceFailureFlag [name\_of\_service]

SimpleSC::GetServiceDelayedAutoStartInfo [name\_of\_service]

SimpleSC::SetServiceDescription [name\_of\_service] [service\_description]

SimpleSC::SetServiceStartType [name\_of\_service] [start\_type]

SimpleSC::SetServiceBinaryPath [name\_of\_service] [binary\_path]

SimpleSC::SetServiceLogon [name\_of\_service] [account] [password]

SimpleSC::SetServiceFailure [name\_of\_service] [reset\_period] [reboot\_message] [command]

[action\_type\_1] [action\_delay\_1] [action\_type\_2] [action\_delay\_2] [action\_type\_3]

[action\_delay\_3]

SimpleSC::SetServiceFailureFlag [name\_of\_service] [failure\_actions\_on\_non\_crash\_failures]

SimpleSC::SetServiceDelayedAutoStartInfo [name\_of\_service] [delayed\_autostart]

SimpleSC::GrantServiceLogonPrivilege [account]

SimpleSC::RemoveServiceLogonPrivilege [account]

SimpleSC::ServiceIsPaused [name\_of\_service]

SimpleSC::ServiceIsRunning [name\_of\_service]

SimpleSC::ServiceIsStopped [name\_of\_service]

SimpleSC::GetErrorMessage [error\_code]

Для установки в качестве службы заменим в скрипте существующую секцию на следующую:

Section

; Remove a service

SimpleSC::RemoveService "Debugger\_service"

Pop $0 ; returns an errorcode (<>0) otherwise success (0)

SectionEnd

Результат приведен на рисунке 4.

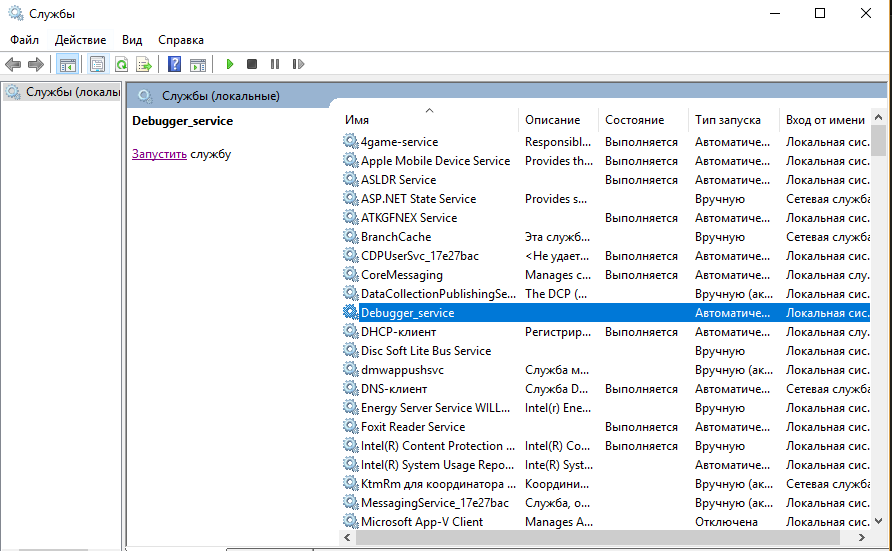


Рис. 4. Установленная служба в списке служб

**Создание инсталлируемой версии в ОС Linux**

Рассмотрим процесс создания пакета для Linux debian 3.16.0-4-586. Пакеты формата –deb позволяют устанавливать в систему различные программы и библиотеки как в виде бинарных файлов, так и из исходных файлов.

Для начала необходимо создать нужные папки:

user@debian:~/debugger$ ls -lR

.:

итого 8

drwxr-xr-x 2 user user 4096 апр 4 10:20 DEBIAN

drwxr-xr-x 3 user user 4096 апр 4 10:19 usr

./DEBIAN:

итого 4

-rw-r--r-- 1 user user 123 апр 4 10:25 control

./usr:

итого 4

drwxr-xr-x 2 user user 4096 апр 4 10:19 bin ./usr/bin:

итого 24

-rwxr-xr-x 1 user user 21084 апр 4 10:19 SimpleDebugger

Папка DEBIAN содержит файлы, используемые при установке. А папка usr содержит инсталлируемые файлы в том порядке, в котором они будут скопированы в систему. В нашем случае это один исполняемый файл, который находится в папке bin.

Папка DEBIAN может содержать следующие файлы:

* **обязательный** DEBIAN/control
* DEBIAN/copyright: © / лицензия
* DEBIAN/changelog: история изменений
* DEBIAN/rules: правила компиляции
* DEBIAN/conffiles: список файлов конфигурации
* DEBIAN/dirs: список папок для создания
* DEBIAN/md5sums: контрольные суммы файлов
* DEBIAN/menu: создание пунктов меню
* DEBIAN/watch: мониторинг сайта, откуда была скачана программа
* DEBIAN/inid.d: init-скрипт

В нашем примере создадим только обязательный файл control. Control — центральный файл пакета, описывающего все основные свойства. Файл — текстовый, состоящий из пар «Атрибут: значение».

В таблице приведены обязельные поля, определённые для контрольного файла.

Таблица 1. Обязательные поля для контрольного файла

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Описание** | **Примеры** |
| **— основные —** | | |
| **Package:** | Имя пакета: [a-zA-Z0-9-] — только латиница, цифры, и дефис. Имя используется при установке: apt-get install <package> | Package: supersh |
| **Version:** | Версия пакета (и проги внутри). Используется для определения «обновлять ли». | Version: 1.0-1<br>Version: 2009.12.12-1 |
| Формат принят такой: <версия\_программы>-<версия\_пакета>. |
| Рекомендую **всегда** указывать версию пакета: при изменении структуры пакета цифра увеличивается на единичку. |
| Допустимые символы достаточно вольные: можно использовать дату и буквы. Примеры смотрите сегодня в своём репозитории :) |
| **Maintainer** | Имя и почта мэйнтейнера пакета: человека, который «дебианизировал» приложение. | Maintainer: o\_O Tync <o-o-tync.habrahabr.ru> |
| Формат произвольный, но принято имя <e-mail> |
| **Architecture** | Архитектура процессора, для которой предназначен пакет. | Architecture: all |
| Допустимые значения: i386, amd64, all, source |
| all используется для скриптов: они же портативные, верно? :) |
| source используется для компилируемых пакетов с исходниками |
| **Section** | Определяет задачу, для которой приложение обычно используется (группа приложений). | Section: misc |
| Возможные значения: admin, base, comm, contrib, devel, doc, editors, electronics, embedded, games, gnome, graphics, hamradio, interpreters, kde, libs, libdevel, mail, math, misc, net, news, non-free, oldlibs, otherosfs, perl, python, science, shells, sound, tex, text, utils, web, x11 |
| **Description** | Описание пакета. | Description: Short.<br>␣Long <br>␣goes here.<br>␣.<br>␣New line. |
| Описание состоит из двух частей: короткое описание (70 символов) на той же строке, и длинное описание на последующих строках, **начинающихся с пробела**. |
| В расширенном описании все переводы строки игнорируются. Для вставки \n используется одиночная точка. |

В итоге файл control, созданный для проекта debugger выглядит следующим образом:

user@debian:~/debugger$ cat DEBIAN/control

Package: debugger

Version: 1.0

Maintainer: Danil <danil@ru>

Architecture: all

Section: utils

Description: Debugger for C++

Сборка пакета:

user@debian:~$ dpkg-deb --build debugger

dpkg-deb: сборка пакета «debugger» в файл «debugger.deb».

Установка:

root@debian:/home/user# dpkg -i debugger.deb

(Чтение базы данных … на данный момент установлен 212461 файл и каталог.)

Подготовка к распаковке debugger.deb …

Распаковывается debugger (1.0) на замену (1.0) …

Настраивается пакет debugger (1.0) …

Проверка:

root@debian:/home/user# SimpleDebugger

In debugger process 1581

Exit from debugging cycle

Program exited with status (0)

В других версиях Linux существуют свои утилиты для создания пакетов. Например в Arch для этого предназначена The Arch Build System. Вся информация для создания пакета помещена в файл PKGBUILD. Когда вы запускаете makepkg, он ищет PKGBUILD в текущей рабочей директории и затем собирает приложение из исходного кода, следуя инструкциям в PKGBUILD. После успешной компиляции полученные бинарные файлы, как и вся необходимая метаинформация (такие как версия пакета и зависимости), архивируются в пакетный файл имя.pkg.tar.gz, который легко может быть установлен при помощи команды pacman -Up <имя\_файла\_пакета>.

**Создание демона в ОС Linux**

Демон — компьютерная программа в системах класса UNIX, запускаемая самой системой и работающая в фоновом режиме без прямого взаимодействия с пользователем. Для создания демона необходимо создать несколько функций, основная из которых – функция мониторинга. Основная цель мониторинга — отслеживание состояния процесса демона.

Весь мониторинг работы демона будет заключен в функцию MonitorProc. Весь смысл мониторинга заключается в том, чтобы запустить дочерний процесс и следить за ним, и в зависимости от кода его завершения, перезапускать его или завершать свою работу.

Исходный код функции мониторинга:

int monitorProcess() {

int pid;

int status;

sigset\_t sigset;

siginfo\_t siginfo;

// настраиваем сигналы которые будем обрабатывать

sigemptyset(&sigset);

// сигнал остановки процесса пользователем

sigaddset(&sigset, SIGQUIT);

// сигнал для остановки процесса пользователем с терминала

sigaddset(&sigset, SIGINT);

// сигнал запроса завершения процесса

sigaddset(&sigset, SIGTERM);

// сигнал посылаемый при изменении статуса дочернего процесса

sigaddset(&sigset, SIGCHLD);

sigprocmask(SIG\_BLOCK, &sigset, NULL);

// данная функция создаст файл с нашим PID'ом

setPidFile(PID\_FILE);

// бесконечный цикл работы

for (;;) {

// создаём потомка

pid = fork();

if (pid == -1) {

break;

} else if (!pid) {// если мы потомок

// данный код выполняется в потомке

// запустим функцию отвечающую за работу демона

status = workingProcess();

// завершим процесс

exit(status);

} else {// если мы родитель

// данный код выполняется в родителе

// ожидаем поступление сигнала

sigwaitinfo(&sigset, &siginfo);

// если пришел какой-либо ожидаемый сигнал

// убьем потомка

kill(pid, SIGKILL);

status = 0;

break;

}

}

// удалим файл с PID'ом

unlink(PID\_FILE.c\_str());

return status;

}

Исходный код демона:

int workingProcess() {

for(;;) {

std::ofstream file;

file.open("/tmp/Debugger.txt");

file << "Debugger\n";

file.close();

sleep(10);

}

return 0;

}

Функция создания демона:

int main(int argc, char\*\* argv) {

int status;

int pid;

// создаем потомка

pid = fork();

if (pid == -1)

std::cout << "Error” << std::endl;

return -1;

} else if (!pid) {// если это потомок

// данный код уже выполняется в процессе потомка

// разрешаем выставлять все биты прав на создаваемые файлы,

// иначе у нас могут быть проблемы с правами доступа

umask(0);

// создаём новый сеанс, чтобы не зависеть от родителя

setsid();

// переходим в корень диска, если мы этого не сделаем, то могут быть проблемы.

// к примеру с размантированием дисков

chdir("/");

// закрываем дискрипторы ввода/вывода/ошибок, так как нам они больше не понадобятся

close(STDIN\_FILENO);

close(STDOUT\_FILENO);

close(STDERR\_FILENO);

// Данная функция будет осуществлять слежение за процессом

status = monitorProcess();

return status;

}

else {// если это родитель

// завершим процес, т.к. основную свою задачу (запуск демона) мы выполнили

return 0;

}

}

Тестирование:

user@debian:~$ g++ -o debug daemon.cpp

user@debian:~$ ./debug

user@debian:~$ ps -xf

PID TTY STAT TIME COMMAND

1173 ? SNl 0:00 \\_ /usr/lib/tracker/tracker-miner-apps

1187 ? SNl 0:00 \\_ /usr/lib/tracker/tracker-miner-user-guides

**1948 ? Ss 0:00 ./debug**

**1949 ? S 0:00 \\_ ./debug**

1737 ? Sl 0:00 /usr/lib/dleyna-renderer/dleyna-renderer-service

1713 ? Sl 0:00 /usr/lib/gvfs/gvfsd-burn --spawner :1.9 /org/gtk/gvfs

user@debian:~$ cat /tmp/Debugger.txt

Debugger

**Вывод**

В данной работе мы познакомились с возможностями ОС Windows и Linux для создания и установки программ. Сначала были изучены основы создания инсталляторов для Windows. Пакет NSIS обладает широкой функциональностью в плане создания инсталляторов и использует удобный, легкий и понятный скриптовый язык для их создания. С помощью плагинов NSIS позволяет устанавливать программы как службы.

Далее были изучены основы создания установочных пакетов для Ubuntu. Изученная система позволяет создавать пакеты как из бинарных файлов, так и компилировать эти бинарные файлы из исходных файлов во время установки. Создание демона для ОС Linux производилось с помощью кода и не вызвало трудностей.

**Список использованных источников**

1. Интернет источник. <http://kav-soft.ru/onlineHelp/NSIS/source/Article/1.html>. Дата обращения: 04.04.2017
2. Интернет источник. <http://nsis.sourceforge.net/NSIS_Simple_Service_Plugin> Дата обращения: 04.04.2017
3. Интернет источник. <https://habrahabr.ru/post/129207/> Дата обращения: 04.04.2017
4. Интернет источник. <https://habrahabr.ru/post/78094/> Дата обращения: 04.04.2017
5. Интернет источник. <https://pastebin.com/jdX5wn0E> Дата обращения: 04.04.2017
6. Интернет источник. <https://wiki.archlinux.org/index.php/Creating_packages_(%D0%A0%D1%83%D1%81%D1%81%D0%BA%D0%B8%D0%B9)> Дата обращения: 04.04.2017