

Параметры и возвращаемое значение функции main

Все программы, написанные на C, содержат функцию main. При запуске программы это будет первая пользовательская функция, исполняемая в контексте процесса (но не первая функция вообще, так как до main будут выполнены процедуры инициализации, зависящие от операционной системы и используемых библиотек C).

Функция main получает параметры от операционной системы (точнее, от оболочки (shell), из которой запускается программа). Возвращаемое значение функции main считывается оболочкой и в командной строке доступно через переменную \$?.

Внимание! Через \$? доступен только младший байт возвращаемого значения main

Есть несколько определений функции main в библиотеке C:

int main() – сокращенное определение без аргументов

int main(int argc, char *argv[]) – стандартное определение

int main(int argc, char *argv[], char *envp[]) – расширенное определение, не входит в стандарт ANSI C, доступно в UNIX, UNIX-подобных ОС и Windows.

Здесь

argc – число аргументов, переданное в командной строке при запуске программы

char *argv[] – массив указателей на строки со значениями аргументов командной строки, содержит argc элементов

char *envp[] – массив указателей на строки с именами и значениями переменных окружения в виде ИМЯ=ЗНАЧЕНИЕ (подробнее о стандартных переменных окружения UNIX можно посмотреть здесь: https://www.gnu.org/software/libc/manual/html_node/Standard-Environment.html). Число переменных окружения не передается в main. Массив envp[] нужно читать последовательно с начала, пока не будет прочитано значение NULL.

Переменные окружения можно устанавливать из командной строки командой export:

export MY_ENV_VAR="Моя переменная окружения"

Компиляция программы из командной строки

В UNIX и UNIX-подобных системах обычно используются компиляторы GNU. Запускается командой gcc (сокращение от Gnu Compilers Collection). Нужный компилятор выбирается по расширению файла, поэтому для программы на C расширение .c является обязательным. Опция -o позволяет задать имя выходного файла.

Пример:

```
admin@ubuntu-for-os-study: ~/Software/processes
File Edit View Search Terminal Help
admin@ubuntu-for-os-study:~/Software/processes$ cat hello.c
#include <stdio.h>

int main()
{
    printf("Hello World!\n");
    return 0;
}

admin@ubuntu-for-os-study:~/Software/processes$ gcc hello.c -o hello
admin@ubuntu-for-os-study:~/Software/processes$ ./hello
Hello World!
admin@ubuntu-for-os-study:~/Software/processes$
```

Разбор аргументов командной строки

Существует общепринятый синтаксис командной строки, позволяющий задавать опции работы программы. Аргументы командной строки разделяются пробелами (если значение параметра включает пробел, то такое значение нужно взять в кавычки). Синтаксис опции следующий:

-<код опции>[значение опции]

Опцией является аргумент командной строки, начинающийся с символа '-'. Опция может находиться в произвольном аргументе командной строки (не обязательно опции должны начинаться с первого аргумента, не обязательно опции должны идти в командной строке подряд (между ними могут быть аргументы, которые опциями не являются), после опций могут быть указаны еще аргументы).

Специальная опция '--' принудительно завершает список опций, даже если в командной строке есть еще аргументы.

Если опция предполагает установку значения, то его нужно указать после ключа (значение можно отделить от ключа пробелом, но это не обязательно).

Опции без параметров можно указывать без пробела за одним '-', например, ls -l -a -i и ls -lai приведут к одинаковому результату.

Порядок следования опций не имеет значения, например, ls -lai ls -ail ls -lia приведут к одинаковому результату.

В библиотеке C существует специальная функция getopt для работы с опциями (определена в unistd.h):

```
int getopt(int argc, char * const argv[], const char *optstring)
```

В функцию нужно передать аргументы main и строку опций, определяющую список опций. Символ ':' после опции означает наличие у опции значения, два двоеточия '::' означает, что значение является опциональным и может не указываться. Например "ab:c::" - определены три опции: -a, -b и -c. Опция -a используется без значения, опция -b требует обязательно указать значение, опция -c может использоваться как со значением, так и без значения.

Если значение опции должно передаваться обязательно, то любая строка после кода опции будет трактоваться как значение (даже если она начинается с '-'). Если значение опции может передаваться опционально, то любая строка после кода опции, введенная без пробела после кода

опции, будет трактоваться как значение (даже если она начинается с '-'). Если же ввести пробел после кода опции, то следующая строка будет трактоваться как параметр, только если она не распознается как опция.

Функция getopt использует глобальные переменные для управления и возвращения значений параметров опций:

optarg - указатель на строку со значением параметра опции или NULL, если нет параметра

optind - индекс текущей опции в массиве argv. Если в строке параметров встретится специальная опция '--', то в переменной optind будет индекс аргумента, следующего за '--'

Документацию по getopt см. <https://man7.org/linux/man-pages/man3/getopt.3.html> или man 3 getopt в командной строке.

Пример:

```
#include <unistd.h>
```

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    while (1)
```

```
    {
```

```
        int c = getopt(argc, argv, "ab:c::d");
```

```
        if (c == -1)
```

```
        {
```

```
            break;
```

```
        }
```

```
        switch (c)
```

```
        {
```

```
            case 'a':
```

```
                printf("option 'a'\n");
```

```
                break;
```

```
            case 'b':
```

```
                printf("option 'b' with '%s'\n", optarg);
```

```
                break;
```

```
            case 'c':
```

```
                if (optarg)
```

```
                {
```

```

        printf("option 'c' with '%s'\n", optarg);
    }
    else
    {
        printf("option 'c' without argument\n");
    }
    break;
case 'd':
    printf("option 'd'\n");
    break;
}
}
return 0;
}

```

Длинные опции

Традиционные опции обозначаются одним символом (код опции), что не всегда удобно при работе (трудно запомнить названия однобуквенных кодов), удобнее было бы обозначать коды опций осмысленными словами. Такую возможность дает функция `getopt_long` (определена в `getopt.h`)

```
int getopt_long(int argc, char * const argv[], const char *optstring, const struct option *longopts,
               int *longindex);
```

Длинные опции начинаются с двух дефисов '--'.

В функцию `getopt_long` нужно передать аргументы `main`. Строка опций `optstring` аналогична рассмотренной в `getopt`. Массив `longopts` описывает длинные опции.

Элементами массива `longopts` являются экземпляры структуры `option`, определенной следующим образом:

```
struct option {
    const char *name; // название опции
    int has_arg; // наличие аргумента: no_argument, required_argument, optional_argument
    int *flag; // =NULL
    int val; // возвращаемое значение getopt_long, если будет найдена данная опция
};
```

Название опции может содержать пробелы, но тогда при использовании опции ее нужно писать в кавычках.

Последний элемент массива `longopts` должен быть заполнен нулями (все поля структуры `option` нужно обнулить).

Параметр `longindex` может быть `NULL`

Пример использования getopt_long:

```
#include <getopt.h>
```

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    struct option longopts[] =
```

```
    {
```

```
        {
```

```
            .name = "advanced",
```

```
            .has_arg = no_argument,
```

```
            .flag = NULL,
```

```
            .val = 'a'
```

```
        },
```

```
        {
```

```
            .name = "base address",
```

```
            .has_arg = required_argument,
```

```
            .flag = NULL,
```

```
            .val = 'b'
```

```
        },
```

```
        {
```

```
            .name = "cursor",
```

```
            .has_arg = required_argument,
```

```
            .flag = NULL,
```

```
            .val = 'c'
```

```
        },
```

```
        {
```

```
            .name = "debug",
```

```
            .has_arg = no_argument,
```

```
            .flag = NULL,
```

```
            .val = 'd'
```

```
        },
```

```

        {
        }
};
while (1)
{
    int c = getopt_long(argc, argv, "ab:c::d", longopts, NULL);
    if (c == -1)
    {
        break;
    }
    switch (c)
    {
        case 'a':
            printf("option 'a'\n");
            break;
        case 'b':
            printf("option 'b' with '%s'\n", optarg);
            break;
        case 'c':
            if (optarg)
            {
                printf("option 'c' with '%s'\n", optarg);
            }
            else
            {
                printf("option 'c' without argument\n");
            }
            break;
        case 'd':
            printf("option 'd'\n");
            break;
    }
}

```

```

    }

    return 0;
}

```

Создание нового процесса

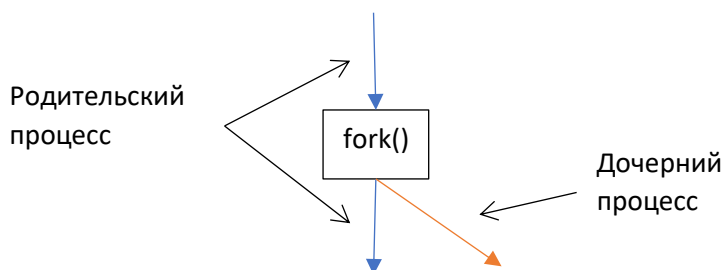
В UNIX и UNIX-подобных системах новый процесс создается вызовом `fork()`, который создает дочерний процесс на основе копии родительского процесса с незначительными модификациями в памяти дочернего процесса. Основные различия дочернего и родительского процесса после завершения `fork` будут следующие¹:

- Дочерний процесс получит собственный уникальный идентификатор процесса (PID)
- Идентификаторы родительского процесса (PPID) в структурах данных дочернего и родительского процесса будут различаться
- Статистика работы дочернего процесса обнуляется
- Дочерним процессом не наследуются сигналы, ожидающие доставки (сигналы, полученные родительским процессом, но еще не обработанные им)
- Разное возвращаемое значение `fork()` (в дочернем процессе `fork()` возвращает ноль, в родительском процессе `fork()` возвращает PID дочернего процесса или -1 в случае ошибки)

Библиотечная функция `fork()` объявлена в заголовочном файле `unistd.h`

```
pid_t fork();
```

После завершения функции `fork()` родительский и дочерний процессы будут выполнять одну и ту же программу с момента завершения `fork()`, т.е. нить выполнения процесса как-бы разветвляется в функции `fork()`².



// до вызова `fork()` код выполняется только в родительском процессе

```
pid_t pid = fork();
```

```
if (pid == 0)
```

```
{
```

// этот блок кода будет выполняться только в дочернем процессе

```
}
```

```
else if (pid == -1)
```

```
{
```

// этот блок кода будет выполняться только в родительском процессе

// в случае ошибки в `fork()`

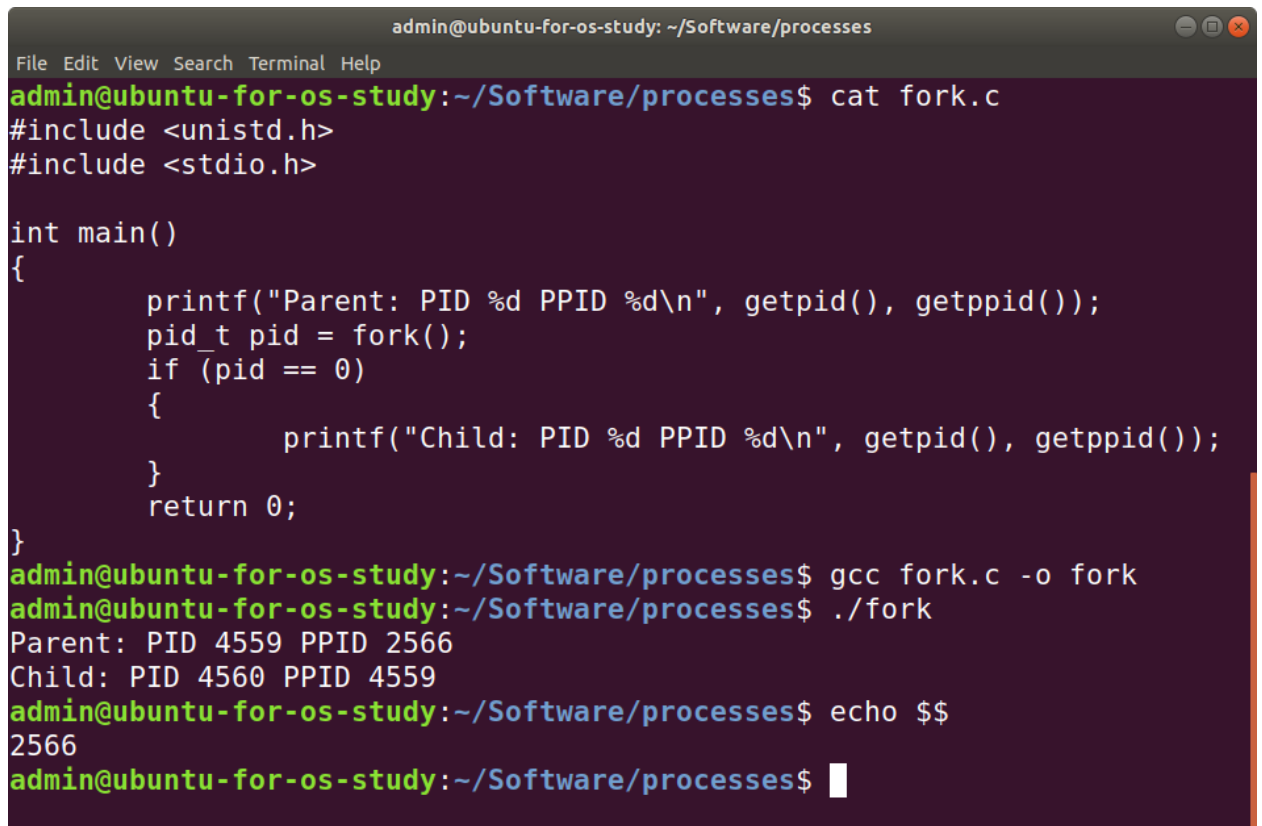
¹ В конкретных операционных системах могут быть специфические отличия, связанные, например, с наследованием состояния семафоров или наследованием блокировок физической памяти.

² В переводе с англ. `fork` – развилка, разветвление, ответвление.

```
}
```

// далее код будет независимо выполняться и в родительском и в дочернем процессе

Пример – вывод PID родительского и дочернего процессов.



```
admin@ubuntu-for-os-study: ~/Software/processes
File Edit View Search Terminal Help
admin@ubuntu-for-os-study:~/Software/processes$ cat fork.c
#include <unistd.h>
#include <stdio.h>

int main()
{
    printf("Parent: PID %d PPID %d\n", getpid(), getppid());
    pid_t pid = fork();
    if (pid == 0)
    {
        printf("Child: PID %d PPID %d\n", getpid(), getppid());
    }
    return 0;
}
admin@ubuntu-for-os-study:~/Software/processes$ gcc fork.c -o fork
admin@ubuntu-for-os-study:~/Software/processes$ ./fork
Parent: PID 4559 PPID 2566
Child: PID 4560 PPID 4559
admin@ubuntu-for-os-study:~/Software/processes$ echo $$
2566
admin@ubuntu-for-os-study:~/Software/processes$
```

Видно, что PID родительского процесса (4559) совпадает с PPID дочернего. Кроме того, родительским процессом для процесса, исполняющего нашу программу, является командная оболочка (PID 2566).

Ожидание завершения дочерних процессов

Родительский процесс может дожидаться завершения дочернего процесса и получить его код возврата (возвращаемое значение `main` дочернего процесса). Для ожидания завершения дочернего процесса используется вызов `wait`, объявленный в заголовочном файле `sys/wait.h`

```
pid_t wait(int *status);
```

Вызов `wait` приостанавливает выполнение вызывающего процесса до завершения любого из его дочерних процессов. Функция `wait` возвращает PID завершившегося процесса. Параметр `status` является опциональным (можно передать `NULL`), но если передан ненулевой указатель на целочисленную переменную, то в этой переменной будет сохранен код возврата завершеного дочернего процесса.


```
admin@ubuntu-for-os-study: ~/Software/processes
File Edit View Search Terminal Help
admin@ubuntu-for-os-study:~/Software/processes$ cat wait.c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main()
{
    pid_t pid = fork();
    if (pid == 0)
    {
        sleep(3);
        return 0;
    }
    printf("waiting for child %d ...\n", pid);
    int code;
    pid_t cpid = wait(&code);
    printf("child %d is done with code %d\n", cpid, code);
    return 0;
}

admin@ubuntu-for-os-study:~/Software/processes$ gcc wait.c -o wait
admin@ubuntu-for-os-study:~/Software/processes$ ./wait
waiting for child 4652 ...
child 4652 is done with code 0
admin@ubuntu-for-os-study:~/Software/processes$
```

Замена программы процесса

В большинстве случаев, дочерние процессы, дублирующие родительский процесс, не нужны. Механизм `fork` используется для создания нового процесса, который затем заменяет унаследованную родительскую программу на новую вызовом `execve`, объявленном в заголовочном файле `unistd.h`.

`int execve(const char *filename, char *const argv [], char *const envp[]);`

`filename` – файл исполняемой программы (если программа не находится в текущем каталоге, то необходимо указать путь к файлу программы)

`argv` и `envp` будут переданы в функцию `main` новой программы.

При успешном выполнении, функция `execve` не возвращает значений (и вообще не завершается в контексте вызывающей программы) – управление передается новой программе. Новая программа работает в контексте того же процесса, что и старая программа, т.е. PID поле вызова `execve` сохраняется.

Для удобства использования, в `unistd` определено еще несколько вариантов функции `exec`, отличающиеся набором и способом передачи параметров:

`int execl(const char *path, const char *arg, ...);`

`int execlp(const char *file, const char *arg, ...);`

`int execlxe(const char *path, const char *arg , ..., char * const envp[]);`

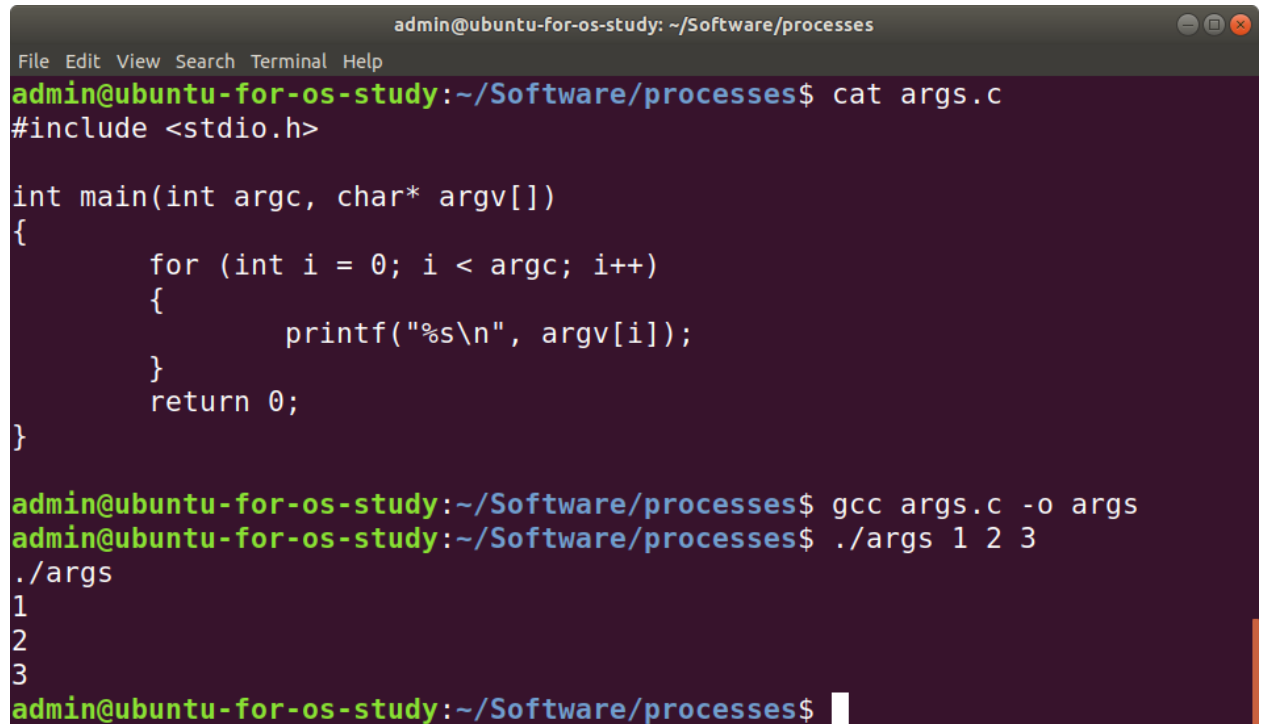
`int execv(const char *path, char *const argv[]);`

```
int execvp(const char *file, char *const argv[]);
```

Поэтому часто функцию замены программы называют просто ехес, не уточняя, о какой именно ее реализации идет речь. Подробнее см.

<https://www.opennet.ru/man.shtml?topic=exec&category=3&russian=0>

Для иллюстрации использования ехес сначала подготовим небольшую программу, которая просто выведет на экран переданные ей аргументы.

A terminal window titled 'admin@ubuntu-for-os-study: ~/Software/processes' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
admin@ubuntu-for-os-study:~/Software/processes$ cat args.c
#include <stdio.h>

int main(int argc, char* argv[])
{
    for (int i = 0; i < argc; i++)
    {
        printf("%s\n", argv[i]);
    }
    return 0;
}

admin@ubuntu-for-os-study:~/Software/processes$ gcc args.c -o args
admin@ubuntu-for-os-study:~/Software/processes$ ./args 1 2 3
./args
1
2
3
admin@ubuntu-for-os-study:~/Software/processes$
```

Затем напомним основную программу с использованием функции ехесl, позволяющей просто перечислить аргументы новой программы через запятую.

```
admin@ubuntu-for-os-study: ~/Software/processes
File Edit View Search Terminal Help
admin@ubuntu-for-os-study:~/Software/processes$ cat exec.c
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <stdio.h>

int main()
{
    pid_t pid = fork();
    if (pid == 0)
    {
        execl("args", "1", "2", "3", NULL);
        perror("execl");
    }
    wait(NULL);
    return 0;
}
admin@ubuntu-for-os-study:~/Software/processes$ gcc exec.c -o exec
admin@ubuntu-for-os-study:~/Software/processes$ ./exec
1
2
3
admin@ubuntu-for-os-study:~/Software/processes$
```

В программе `exec.c` функция `perror` выведет подробные сведения об ошибке, если функция `execl` не выполнится.

Ожидание завершения дочернего процесса (вызов `wait`) не является обязательным и сделан здесь только для красоты вывода программы в консоль. Если не ждать завершения дочернего процесса, то после завершения родительского процесса (исполняет программу `exec.c`) консоль выведет новое приглашение, и дочерний процесс (исполняет программу `args.c`) сделает свой вывод после приглашения, что ухудшит восприятие результата работы программы.

Также обратите внимание на различие в выводе программы `args.c` при ее вызове из командной строки и из программы `exec.c`. При вызове из командной строки `argv[0]` в функции `main` программы `args.c` содержит имя файла программы, а при вызове `args.c` из `exec.c` – нет.

Имя файла программы оказывается в `argv[0]` в функции `main` только потому, что его туда явно заносит вызывающая программа. В нашем случае, чтобы сохранить общепринятый набор `argv`, нужно написать `execl("args", args, "1", "2", "3", NULL);`

Работа с файлами и переопределение стандартных потоков ввода-вывода

Для чтения и записи файлов используются вызовы `read` и `write` соответственно, определенные в `unistd`

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

В качестве аргументов в функции передаются файловый дескриптор и буфер данных. Параметр `count` содержит размер блока данных для чтения или записи. Функции возвращают число прочитанных или записанных байтов данных или `-1` при ошибке.

Файловые дескрипторы программы получают из вызова `open`, объявленного в заголовочном файле `fcntl.h`

```
int open(const char *pathname, int flags, mode_t mode);
```

Здесь `pathname` – имя файла, `flags` определяют поведение функции, например, флаг `O_CREAT` требует создать файл, если он не существует, флаг `O_TRUNC` требует сбросить содержимое существующего файла и т.п., см. <https://www.opennet.ru/cgi-bin/opennet/man.cgi?topic=open&category=2>. Параметр `mode` определяет права доступа к файлу и используется, только если указан флаг `O_CREAT` (во всех остальных случаях игнорируется, для удобства есть объявление функции `int open(const char *pathname, int flags)` без параметра `mode`).

Функция возвращает дескриптор открытого файла или -1 при ошибке.

Пример:

```
Int fd = open("my file", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

Открывает файл "my file" в текущем каталоге для записи. Если файл не существует, то создает его с правами доступа на чтение для всех и группы-владельца, и с правами чтения и записи для владельца файла.

Закрытие ненужных файловых дескрипторов выполняется вызовом `close`, определенном в заголовочном файле `unistd.h`

```
int close(int fd);
```

Важно! Функция `open` выбирает первый свободный файловый дескриптор в таблице файловых дескрипторов процесса.

С учетом этого свойства, можно легко переопределить файловые дескрипторы стандартных потоков ввода-вывода. Например:

```
close(2)
```

```
open("my file", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

перенаправляет стандартный вывод сообщений об ошибках в файл "my file".

Для определенности, если для файла требуется использовать конкретный дескриптор, например, для перенаправления стандартных потоков ввода-вывода, лучше использовать вызов `dup2`, определенный в `unistd.h`

```
int dup2(int oldfd, int newfd);
```

Функция связывает `newfd` с тем же файлом, с которым связан `oldfd`. Если с `newfd` уже был связан файл, то этот файл автоматически закрывается. Таким образом, после вызова `dup2` будет два файловых дескриптора, сопоставленных с одним и тем же файлом.

Например, следующий код перенаправляет стандартный вывод сообщений об ошибках в файл "my file".

```
int tmp = open("my file", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

```
dup2(tmp, 2); // теперь файловый дескриптор 2 (stderr) связан с файлом "my file"
```

```
close(tmp); // лишний дескриптор закрываем
```