СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Опорный конспект по теме №3

«Переменные окружения. Аргументы командной строки»

1. Переменные окружения

Переменные окружения в операционных системах задаются во время начала сеанса пользователя и необходимы для выставления определенных системных значений, использующихся в прикладных программах.

Начальные переменные окружения задаются из файлов `/etc/environment` и `/etc/profile`. Получение переменных из этих файлов не зависит от того, от чьего имени запущен сеанс, пользовательские переменные задаются в файлах Shell RC (что предположительно расшифровывается как *run commands*), например `~/.bashrc` относится к конфигурационному файлу командной оболочки Bash.

Во всех файлах определяются как системные переменные, так и некоторые пользовательские. Синтаксис определения переменной имеет следующий вид:

```
{NAME}=[VALUE]
```

где `{NAME}` — произвольное наименование переменной, как правило, записанное в верхнем регистре, `[VALUE]` — значение в виде строки (может быть пустым). Заметьте, что между именем и значением не должно быть пробелов, отделяющих знак равно, — это синтаксическая ошибка.

Существует набор системных переменных окружения с определенной функцией:

- `PATH` — переменная, хранящая набор разделенных двоеточием путей директорий, которые командная оболочка просматривает в процессе поиска некоторого имени, записанного не как путь. Например, когда командная оболочка интерпретирует команду

```
ls -l /home/user
```

для нее будет неизвестно, что такое `ls`, так как это имя не представляет собой путь, поэтому командная оболочка будет пытаться найти некоторый файл с именем `ls` в директориях, указанных в переменной PATH.
- `SHELL` — путь до исполняемого файла используемой командной оболочки.
- `HOME` — путь до домашней директории пользователя сеанса.
- `USER` — имя текущего пользователя.
- `TERM` — путь до эмулятора терминала.

Некоторые переменные могут быть заданы для использования в различных прикладных программах, и они уже считаются пользовательскими переменными окружения:

- `EDITOR` или `VISUAL` — путь до текстового редактора при использовании команд edit и sudoedit.

- LD_LIBRARY_PATH — пути библиотек для компоновщика GNU Linker (ld) и программного загрузчика.
- IFS — внутренний разделитель полей (сокр. от *Internal Field Separator*) при раскрытии некоторых шаблонов в Bash (цикл for, оператор [@] для списков).
- WINEPREFIX — путь до корневой директории окружения компонентов слоя совместимости Wine.
- DISPLAY — путь до устройства/сервера, на котором необходимо отображать графические примитивы (используется в сервере окон X.Org).

Переменные можно задавать самостоятельно прямо в командной оболочке. Для передачи переменных во внешние приложения (дочерние, запущенные из командной оболочки) в Bash используется команда

```
export {NAME}[={VALUE}]
```

причем если значение VALUE не задается через равно, то export ищет переменную с именем NAME среди уже созданных и делает ее видимой для дочерних процессов.
Возникает проблема, что данная переменная будет видима для всех дочерних процессов, и если она не нужно, то ее стоит удалить во избежание неправильной работы приложений с помощью команды

```
unset {NAME}
```

Такое поведение может быть задано в одной команде: переменную можно создать только для использования в вызываемом приложении, если ее указать непосредственно перед командой, вызывающей это приложение:

```
{NAME}=[VALUE] {COMMAND}
```

Например,

```
IP=”192.168.18.15” network configure
```

создаст переменную IP только в окружении процесса network, другие процессы данную переменную видеть не будут.

2. Использование переменных окружения в программах на C/C++

При написании программы на C/C++ переменные окружения можно получать с использованием функции getenv из заголовочного файла <cstdlib>.

```
char *getenv(const char *name);
```

Данная функция принимает на вход строку в стиле C, обозначающую имя переменной и возвращает строку в стиле C значения этой переменной. Если переменной не существует, будет возвращен нулевой указатель.
Переменные также можно изменять функцией setenv, удалять с помощью unsetenv и очищать с помощью clearenv, но чаще всего они только считываются в программе и не изменяются, так как их изменение затрагивает только текущее окружение и может передаваться только в другие дочерние процессы.

Пример простого калькулятора представлен ниже. Для работы калькулятора требуется указать три переменные окружения:

- FIRST — первое число
- SECOND — второе число
- OP — операция над числами

```
#include <cstdlib>
#include <string>
#include <iostream>

#define CALC_IF_ACTION(OP) if (op == # OP [0]) { result = f OP s; }

int main()
{
    char* first = getenv("FIRST");
    char* second = getenv("SECOND");
    char* operation = getenv("OP");

    if (first == nullptr || second == nullptr ||
        operation == nullptr || operation[0] == '\0')
    {
        std::cerr << "One or several environment variables FIRST, SECOND or OP are not
set" << std::endl;
        return 1;
    }

    double f, s, result;
    char op = operation[0];

    try
    {
        f = std::stod(first);
        s = std::stod(second);
    }
    catch (...)
    {
        std::cerr << "FIRST or SECOND are not numbers" << std::endl;
        return 1;
    }

    CALC_IF_ACTION(+)
    else CALC_IF_ACTION(-)
    else CALC_IF_ACTION(/)
    else CALC_IF_ACTION(*)
    else
    {
        std::cerr << "Unsupported operation " << op << std::endl;
        return 1;
    }

    std::cout << result << std::endl;
    return 0;
}
```

Вызов калькулятора:

```
user@sysproghost:~/Projects/SampleProject/build/env_calc$ FIRST=2.63 SECOND=7.37 OP=+ ./env_calc
10
user@sysproghost:~/Projects/SampleProject/build/env_calc$ FIRST=2.63 SECOND=7.37 OP=* ./env_calc
19.3831
user@sysproghost:~/Projects/SampleProject/build/env_calc$ FIRST=2.63 SECOND=7.37 OP=/ ./env_calc
0.356852
user@sysproghost:~/Projects/SampleProject/build/env_calc$ FIRST=2.63 SECOND=7.37 OP=% ./env_calc
Unsupported operation %
user@sysproghost:~/Projects/SampleProject/build/env_calc$ FIRST=2.63 SECOND=3.67 ./env_calc
One or several environment variables FIRST, SECOND or OP are not set
user@sysproghost:~/Projects/SampleProject/build/env_calc$
```

Также имеется возможность сразу получить все переменные окружения. Для этого необходимо в коде объявить внешнюю переменную environ типа «массив строк в стиле C» в виде указателя:

```
extern char** environ;
```

Данный массив хранит все переменные, видимые текущим процессом. Количество переменных определить заранее нельзя, поэтому после последней известной переменной окружения идет еще один элемент, являющийся нулевым указателем.

3. Аргументы командной строки

Часто можно увидеть, как выполнение команды сопровождается указанием после имени исполняемого файла некоторых флагов, например

```
ls -l -A -h /home/user
```

запускает исполняемый файл ls с дополнительными флагами `-l` (списочный формат), `-A` (отображать все файлы, включая скрытые, кроме . и ..), `-h` (указывать размеры в человекочитаемых единицах), а также путь, в котором будет работать данная команда `/home/user`.

Все эти дополнительные флаги называются «аргументами команды `ls`». В более общем случае, такие аргументы могут передаваться в любое приложение из командной строки и потому называются *аргументами командной строки*.

В языках C/C++ для получения данных аргументов функция main должна иметь определенную сигнатуру:

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

где `argc` — количество аргументов командной строки, включая сам исполняемый файл (от англ. ***arg****ument* ***c****ount*), а `argv` — вектор (массив) строк в стиле C, хранящий сами аргументы (от англ. ***arg****ument* ***v****ector*). Первый аргумент в `argv` (с индексом 0) — это имя исполняемого файла в том виде, как он был запущен из командной оболочки.

С аргументами можно работать сразу. Например, можно реализовать аналог команды tee[1], которая получает данные из стандартного потока ввода и выводит их как в стандартный поток вывода, так и в некоторый файл:

[1] Название получено от T-образного разветвителя, на английском буква T читается как [ti:], что соответствует английскому написанию «tee»

```cpp
#include <iostream>
#include <fstream>

int main(int argc, char *argv[])
{
    // Если количество аргументов не 2, значит неправильно пользуемся, выводим справку
    if (argc != 2)
    {
        std::cout << "Usage:\nmy_tee {file}\n\n"
                  << "Arguments:\n{file} - filename to duplicate stdin to"
                  << std::endl;
        return 1;
    }

    // Создаем файловый поток, пусть он бросает исключения, если что-то пошло не так
    std::ofstream file;
    file.exceptions(std::ios_base::badbit | std::ios_base::failbit);

    try
    {
        // Открыли файл из аргумента
        file.open(argv[1]);

        std::string line;
        while (std::getline(std::cin, line))
        {
            // Читаем построчно и выводим данные в консоль и в файл
            std::cout << line << std::endl;
            file << line << std::endl;
        }
    }
    catch (const std::exception& ex)
    {
        std::cerr << "Exception occurred: " << ex.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Вызов команды:

```
user@sysproghost:~/Projects/SampleProject/build/env$ echo Hello, world! | ./my_tee hello.txt
Hello, world!
user@sysproghost:~/Projects/SampleProject/build/env$ cat hello.txt
Hello, world!
user@sysproghost:~/Projects/SampleProject/build/env$ echo File in root... | ./my_tee /hello.txt
Exception occurred: basic_ios::clear: iostream error
user@sysproghost:~/Projects/SampleProject/build/env$
```

4. Форматирование аргументов

В программах принято подразделять аргументы по их виду на:

- Короткие аргументы POSIX — записываются через дефис и одну букву (-a).
- Длинный аргументы — записываются через два дефиса и произвольное количество слов в kebab-case (--all-files).

Короткие и длинные аргументы почти всегда выступают как управляющие аргументы, которые часто также называют «ключами». Все остальные аргументы называют позиционными.

Позиционные (BSD формат)	**Короткие (UNIX формат)**	**Длинные (GNU формат)**
Указываются как есть	Указываются с помощью знака дефис и одной латинской буквы	Указываются с помощью двух знаков дефис и латинского слова

При этом согласно спецификации GNU, если в программе используются короткие аргументы, то для каждого из них должен быть эквивалентный длинный аргумент. При этом если задан длинный аргумент, то он необязательно имеет эквивалентный короткий.

Для того, чтобы самостоятельно не пытаться парсить аргументы командной строки, можно пользоваться специальными функциями `getopt` (для коротких аргументов POSIX из заголовочного файла unistd.h) и `getopt_long` (для длинных аргументов GNU из заголовочного файла getopt.h).

Функция `getopt` принимает количество аргументов и вектор аргументов из параметров функции `main`, а также некоторую строку, описывающую возможные аргументы в виде массива букв и специальных символов. Более подробно про парсинг с помощью `getopt` можно прочитать в соответствующей странице man.

Пример парсинга аргументов с помощью `getopt` представлен ниже. Данная программа принимает один аргумент из `-h` (вывести справку), `-r` (прочитать файл), `-w` (записать в файл). При этом если передан аргумент `-w`, то все позиционные аргументы рассматриваются как текст, который нужно записать в файл.

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

namespace Mode
{

using Type = int;

enum : Type
{
    READ = 1,
    WRITE = 2,
    NONE = 0
};

}

// Выводим справку
void printHelp()
```

```
{
    // Не пугаемся R, так строка выведется как есть со всеми символами как есть, включая переносы строк
    std::cout << R"(Usage:
modfile {-r {file}|-w {file} [value]|-h}

Options:
-r - read from {file} to stdin
-w - write to {file} the specified {value}
-h - print this message)" << std::endl;
    return;
}

int main(int argc, char *argv[])
{
    // Если аргумент один, значит программа была запущена без аргументов совсем, выводим справку
    if (argc == 1)
    {
        printHelp();
        return 0;
    }

    std::string file;
    Mode::Type mode = Mode::NONE;

    char opt;
    // Строка опций, расшифровывается как-то так:
    // - допустимые аргументы это h, r и w
    // - r и w требуют обязательного аргумента после них
    const char* optstr = "hr:w:";
    // getopt вернет -1, когда короткие аргументы закончатся
    while ((opt = getopt(argc, argv, optstr)) != -1)
    {
        switch (opt)
        {
            case 'h':
                printHelp();
                return 0;
            case 'r':
                mode |= Mode::READ;
                file = optarg; // Аргумент после r - это файл, он помещается в optarg
                break;
            case 'w':
                mode |= Mode::WRITE;
                file = optarg; // Аргумент после w - это файл, он помещается в optarg
                break;
            case '?':
                // Неизвестный аргумент, сообщение выведет getopt, поэтому сразу завершаем работу программы
                return 1;
        }
    }

    // Проверка на то, что передан только один режим (-r или -w)
    if ((mode & (mode - 1)) != 0)
    {
        std::cout << "Only one argument -r or -w is required" << std::endl;
        return 1;
    }

    if (mode == Mode::READ)
    {
```

```
        // Читаем весь контент из файла
        std::ifstream fs(file);
        std::string line;
        while (std::getline(fs, line))
        {
            std::cout << line << "\n";
        }
        std::cout.flush();
    }
    else if (mode == Mode::WRITE)
    {
        // Удаляем файл, будем перезаписывать его
        unlink(file.c_str());
        std::ofstream fs(file);

        // getopt закинул все позиционные аргументы в конец, они начинаются с позиции
optind
        for (int i = optind; i < argc; ++i)
        {
            fs << argv[i] << " ";
        }
        fs << std::endl;
    }
    else
    {
        std::cerr << "Unexpected error, shutting down!" << std::endl;
        return 1;
    }

    return 0;
}
```

Использование:

```
user@sysproghost:~/Projects/SampleProject/build/env$ ./modfile -w hello.txt "
> Привет!
> Тестируем аргументы командной строки!
> "
user@sysproghost:~/Projects/SampleProject/build/env$ ./modfile -r hello.txt

Привет!
Тестируем аргументы командной строки!

user@sysproghost:~/Projects/SampleProject/build/env$ ./modfile -h
Usage:
modfile {-r {file}|-w {file} {value}|-h}

Options:
-r - read from {file} to stdin
-w - write to {file} the specified {value}
-h - print this message
user@sysproghost:~/Projects/SampleProject/build/env$ ./modfile
Usage:
modfile {-r {file}|-w {file} {value}|-h}

Options:
-r - read from {file} to stdin
-w - write to {file} the specified {value}
-h - print this message
user@sysproghost:~/Projects/SampleProject/build/env$ ./modfile -k
./modfile: invalid option -- 'k'
user@sysproghost:~/Projects/SampleProject/build/env$ ./modfile -r
./modfile: option requires an argument -- 'r'
user@sysproghost:~/Projects/SampleProject/build/env$
```

Примера с `getopt_long` не будет, можете реализовать какую-нибудь программу с ним самостоятельно.

5. Дополнительная информации

Для получения дополнительной информации по переменным окружения и аргументам командной строки воспользуйтесь справочной информацией man и cppreference:

https://man7.org/linux/man-pages/man7/environ.7.html
https://man7.org/linux/man-pages/man3/getenv.3.html
https://man7.org/linux/man-pages/man3/getopt.3.html
https://en.cppreference.com/w/cpp/language/main_function.html