ЧЕРНОВИК (версия 1.1)

1. Работа с текстовыми файлами

Стандартная библиотека ввода/вывода Си – это библиотека буферизированного обмена с файлами и устройствами, которые поддерживает операционная система. К таким устройствам относятся консоль, клавиатура принтеры и многое другое.

Для взаимодействия программы с файлом или устройством библиотека использует тип **`FILE`**, который описывается в заголовочном файле stdio.h. При открытии файла или устройства возвращается указатель на объект этого типа (*файловый указатель*).

fopen

```
FILE* fopen(const char* filename, const char* mode);
```

Функция fopen открывает файл. Она получает два аргумента – строку с именем файла и строку с режимом доступа к файлу. Имя файла может быть как абсолютным, так и относительным. fopen возвращает файловый указатель, с помощью которого далее можно осуществлять доступ к файлу. Если вызов функции fopen прошёл неудачно, то она возвратит NULL.

Некоторые режимы открытия файла.

Режим (mode)	**Описание**
"r"	Чтение. Файл должен существовать.
"w"	Запись. Если файл с таким именем не существует, он будет создан, в противном случае его содержимое будет потеряно.
"a"	Запись в конец файла. Файл создаётся, если не существовал.

Функция fopen может открывать файл в текстовом или бинарном режиме. По умолчанию используется текстовый режим. Если необходимо открыть файл в бинарном режиме, то в конец строки добавляется буква b, например "rb", "wb", "ab".

Замечание

Текстовый файл содержит только печатные символы (т.е. символы которые можно вводить с клавиатуры). Он организован в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’. В конце последней строки этот символ не является обязательным.

Требования операционной системы к символу новой строки в текстовом файле могут быть различными (например, Windows: CR+LF, Linux: LF, Mac: CR). Поэтому чтобы программа была переносимой, символ ‘\n’ преобразуется библиотекой ввода/вывода в представление принятое в конкретной операционной системе.

По этой причине в текстовом файле может не быть однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся в файле.

fclose

```
int fclose(FILE* f);
```

Функция fclose закрывает файл: выполняет запись буферизированных, но еще незаписанных данных, уничтожает непрочитанные буферизированные входные данные, освобождает все автоматически выделенные буфера, после чего закрывает файл или устройство. Возвращает EOF в случае ошибки и нуль в противном случае.

EOF – макрос, который определен в файле stdio.h. Представляет собой отрицательное целое число типа int, которое используется для обозначения конца файла.

Замечание

Операционная система после завершения программы освобождает все ресурсы, которые программа использовала. Поэтому даже если функция fclose не была вызвана, ничего страшного. Зачем же использовать fclose? Можно назвать несколько причин:

1. У программы может быть ограничение на количество одновременно открытых файлов. Если не закрывать неиспользуемые, то может произойти ошибка открытия очередного файла.
2. В Windows когда программа открывает файл, другие программы не могут его открыть или удалить. Поэтому если файл может кому-то понадобиться и, до окончания работы программы ещё далеко, вызывайте fclose.
3. Вывод в файл совершается не сразу, а через вспомогательный буфер. Это сделано для ускорения процесса вывода. Поэтому при закрытии файла выполняется запись буферизованных, но ещё не записанных данных. Если же fclose не была вызвана, то буферизованные данные могут быть потеряны. Поэтому, если вы хотите гарантированно увидеть всё, что вывела программа, то используйте fclose.

fprintf (форматированный вывод)

```
int fprintf(FILE* f, const char* format, ...);
```

Функция fprintf преобразует и выводит данные в файл (или устройство), связанный с файловой переменной f, под управлением строки форматирования format. Возвращает количество записанных символов или, в случае ошибки – EOF.

fscanf (форматированный ввод)

```
int fscanf(FILE* f, const char* format, ...);
```

Функция fscanf считывает данные из файла (или устройства), связанного с файловой переменной f, под управлением строки форматирования format и присваивает преобразованные значения последующим аргументам.

Функция завершает работу, когда «исчерпывается» строка форматирования. При этом она возвращает EOF, если до преобразования очередного значения ей встретился конец файла или появилась ошибка. В противном случае функция возвращает количество введенных значений.

rewind

```
void rewind(FILE* f);
```

Функция rewind позволяет начать обработку файла сначала.

Предопределенные файловые переменные

При инициализации программы библиотека ввода/вывода заводит три файловые переменные *stdin*, *stdout* и *stderr* для:

- Стандартного потока ввода (сокращение от standard input). Программа может читать из него данные.
- Стандартного потока вывода (сокращение от standard output). Программа может выводить в него данные.
- Стандартного потока ошибок (сокращение от standard error). Программа может выводить в него сообщения об ошибках. Это нужно для того, чтобы они не терялись в случае перенаправления stdout.

Обычно стандартные потоки направляются к консоли, но они могут быть перенаправлены операционной системой на другое устройство.

Эти файловые переменные могут использоваться в любой функции, где используется переменная типа FILE*:

```
...

float a, b;

if (scanf("%f%f", &a, &b) != 2)
    printf(stdout, "I/O error\n");
else

...
```

```
...

float a, b;

if (fscanf(stdin, "%f%f", &a, &b) != 2)
    fprintf(stdout, "I/O error\n");
else

...
```

В качестве примера использования вышеперечисленных функций рассмотрим решение следующей задачи. Программа получает на вход файл test.txt, который содержит последовательность из натуральных чисел. Необходимо определить значение наибольшего элемента последовательности.

```
#include <stdio.h>

int get_max(FILE *f, int *max)
{
    int num;

    if (fscanf(f, "%d", max) == 1)
    {
        while (fscanf(f, "%d", &num) == 1)
            if (num > *max)
                *max = num;

        return 0;
    }

    return -1;
}

int main(void)
{
    FILE *f;
    int max;

    f = fopen("test.txt", "r");
```

```
    if (f == NULL)
    {
        printf("I/O error\n");

        return -1;
    }

    if (get_max(f, &max) == 0)
        printf("max is %d\n", max);
    else
        printf("There are not enough data.\n");

    fclose(f);

    return 0;
}
```

2. Параметры командной строки

Материал данного раздела частично приводится по книге С.В. Шапошниковой «Особенности языка Си».

Часто данные передаются в программу из командной строки при ее запуске. Например,

```
# gcc.exe –std=c99 -Wall -Werror -o args.exe args.c
```

Здесь запускается программа gcc.exe, которая из командной строки получает шесть аргументов: -std=c99, -Wall, -Werror, -o, args.exe и args.c.

Если программа написана на языке Си, то после ее запуска управление передается в функцию main, которая и получает аргументы командной строки:

```
int main(int argc, char** argv);
```

Данный вариант функции main получает два параметра:

- целое число (argc), обозначающее количество аргументов (элементов, разделенных пробелами) в командной строке при вызове (следует иметь в виду, что само имя программы также учитывается),
- указатель на массив строк (argv), где каждая строка - это отдельный аргумент из командной строки.

Для приведенного выше примера значение argc равно 7, а массив строк argv определяется как {«gcc.exe», «-std=c99», «-Wall», «-Werror», «-o», «args.exe», «args.c», NULL}.

То, что в программу передаются данные, не означает, что функция main должна их обрабатывать. Если функция main определена без параметров (`int main(void)`), то получить доступ к аргументам командной строки невозможно, но при этом вы можете указывать их при запуске.

Изучите вывод программы, приведенной ниже, запуская ее с различными аргументами командной строки.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("argc = %d\n", argc);

    for (int i = 0; i < argc; i++)
        puts(argv[i]);

    return 0;
}
```

Имена argc и argv не являются обязательными (т.е. вы может использовать любые), но лучше придерживаться именно этих имен, чтобы ваши программы были более понятны не только вам, но и другим программистам.

Чаще всего в программу при запуске передаются имена файлов и «опции» (ключи), которые влияют на процесс выполнения программы. В качестве примера рассмотрим программу, которая выводит на печать последовательность натуральных чисел из текстового файла, имя

которого передается через аргументы командной строки.

```
#include <stdio.h>

void print_nums(FILE *f)
{
    int num;

    while (fscanf(f, "%d", &num) == 1)
        printf("%d\n", num);
}

int main(int argc, char** argv)
{
    FILE *f;

    if (argc != 2)
    {
        fprintf(stderr, "num_reader.exe file-name\n");

        return -1;
    }

    f = fopen(argv[1], "r");
    if (f == NULL)
    {
        fprintf(stderr, "I/O error\n");

        return -2;
    }

    print_nums(f);

    fclose(f);

    return 0;
}
```

3. Обработка ошибочных ситуаций

Информирование об ошибках

Большинство функций стандартной библиотеки при возникновении ошибки возвращают отрицательное число или NULL и записывают в глобальную переменную errno код ошибки. Эта переменная определена в заголовочном файле errno.h. Заметим, что в случае успешного выполнения функции эта переменная просто не изменяется и может содержать любой мусор, поэтому проверять ее имеет смысл лишь в случае, если ошибка действительно произошла.

При использовании функций, устанавливающих errno, можно сверить значение errno со значениями ошибок, определенных в include-файле <errno.h>, или же использовать функции perror и strerror. Если нужно распечатать сообщение о стандартной ошибке - используется perror; если сообщение об ошибке нужно расположить в строке, то используется strerror.

perror

```
void perror(const char *message);
```

Функция perror преобразует значение глобальной переменной errno в строку и записывает эту строку в stderr. Если значение параметра message не равно нулю, то сначала записывается сама строка, за ней ставится двоеточие, а затем следует сообщение об ошибке, определяемое конкретной реализацией.

strerror

```
char* strerror(int errnum);
```

Функция strerror возвращает указатель на строку, содержащую системное сообщение об ошибке, связанной со значением errnum. Эту строку не следует менять ни при каких обстоятельствах.

Обработка ошибок выделения ресурсов

TODO: Подход, который обычно используют студенты

"-" многочисленные места закрытия файлов

TODO: Подход в стиле структурного программирования

"-" вложенность зависит от числа ресурсов

TODO: Подход с использованием goto

"-" требует дисциплины проставления меток

Утверждения

Заголовочный файл assert.h стандартной библиотеки содержит макрос assert для проверки утверждений.

Если утверждение ложно, то макрос выводит информацию о вызове в stderr и вызывает функцию abort. Выводимая информация, включает в себя:

- текст выражения, значение которого равно нулю 0.
- имя файла с исходным кодом;
- строка у файла с исходным кодом.

Для того чтобы отключить проверку утверждений, не обязательно исключать макросы assert из кода программы или комментировать их. Достаточно лишь объявить макрос NDEBUG в программе перед #include <assert.h>.

```
#include <stdio.h>

//#define NDEBUG

#include <assert.h>

long long unsigned int factorial(int n)
{
    long long unsigned int f = 1;

    assert(n >= 0);

    for (int i = 1; i <= n; i++)
       f *= i;

    return f;
}

int main(void)
{
    int num = 3;

    printf("Factorial of %d is %llu\n", num, factorial(num));

    num = 5;
    printf("Factorial of %d is %llu\n", num, factorial(num));

    num = 0;
    printf("Factorial of %d is %llu\n", num, factorial(num));

    num = -10;
    printf("Factorial of %d is %llu\n", num, factorial(num));

    return 0;
}
```

4. Утилита gcov

Утилита *gcov* используется (в том числе) для анализа покрытия кода тестами. Информация, которая получается в результате работы этой утилиты, позволяет ответить на следующие вопросы:

- как часто выполняется каждая строка кода;
- какие строки кода не выполняются;
- какой процент строк кода покрыт тестами (для каждого файла);
- какой процент ветвлений покрыт тестами (для каждого файла).

gcov обрабатывает программы, которые были получены только с помощью компилятора *gcc*. При сборке программы желательно выключить оптимизацию с помощью ключа «-O0» и добавить следующие ключи «-fprofile-arcs» и «-ftest-coverage».

Два последних ключа вынуждают компилятор добавить в исполняемый файл программы дополнительный код, который во время ее выполнения собирает статистическую информацию и сохраняется ее в служебные файлы. *gcov* использует эти файлы для создания «аннотированного» листинга исходного кода программы, который содержит информацию о частоте выполнения каждой строки.

Рассмотрим шаги использования утилиты *gcov* на примере исследования программы из Приложения А.

Шаг 1. Компиляция программы

```
# c99 -Wall -Werror -pedantic -O0 -fprofile-arcs -ftest-coverage sq.c -o sq.exe
```

Шаг 2. Запуск программы
Например, так

```
# sq.exe
Enter a, b, c: 1 2 1
x1 = x2 = -1.000000
```

Шаг 3. Запуск утилиты gcov

Например, так

```
# gcov sq.c
File 'sq.c'
Lines executed:69.23% of 13
Creating 'sq.c.gcov'

File 'c:/mingw/include/stdio.h'
Lines executed:100.00% of 5
Creating 'stdio.h.gcov'
```

Утилита gcov выдаст на экран указанную информацию и создаст «аннотированный» листинг sq.c.gcov (см. Приложение Б). Возможно, что «аннотированный» листинг будет включать в себя несколько файлов.

Изучим содержание «аннотированного» листинга. Он начинается «преамбулой», каждая строка которой имеет вид:

```
-:0:<tag>:<value>
```

Из «преамбулы» нас будет интересовать количество запусков программы:

```
        -:    0:Runs:1
```

У остальных строк «аннотированного» листинга формат следующий:

```
<количество выполнений строки>:<номер строки>:<строка исходного кода>
```

Для строк, которые не содержат код, количество выполнений обозначается символом «-», а для строк, которые не выполнялись ни разу, - «#####».

Повторяя шаги 2 и 3 для разных входных данных и анализируя «аннотированный» листинг, можно добиться полного покрытия кода тестами.

Замечание 1
Следует помнить, что даже 100% покрытие кода тестами не означает, что в программе нет ошибок!

Замечание 2
Ключ «-a» дополнит «аннотированный» листинг информацией о выполнении всех базовых блоков каждой строки программы, а не только основного.

Замечание 3
Ключ «-b» дополнит «аннотированный» листинг информацией статистикой о выполнении условных операторов.

Приложение А

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

int main(void)
{
    float a, b, c, d;

    printf("Enter a, b, c: ");
    if (scanf("%f%f%f", &a, &b, &c) == 3)
    {
        if (a != 0.0)
        {
            d = b * b - 4 * a * c;

            if (d < 0.0)
            {
                printf("There are no real roots\n");
            }
            else if (d > 0)
            {
                printf("x1 = %f, x2 = %f\n", (-b - sqrt(d)) / (2 * a),
                                             (-b + sqrt(d)) / (2 * a));
            }
            else
            {
                printf("x1 = x2 = %f\n", -b / (2 * a));
            }
        }
        else
        {
            printf("Equation is not square\n");
        }
    }
    else
    {
        printf("I/O error\n");
    }

    return 0;
}
```

Приложение Б

```
        -:    0:Source:sq.c
        -:    0:Graph:sq.gcno
        -:    0:Data:sq.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:#include <math.h>
        -:    3:
        1:    4:int main(void)
        -:    5:{
        -:    6:    float a, b, c, d;
        -:    7:
        1:    8:    printf("Enter a, b, c: ");
        1:    9:    if (scanf("%f%f%f", &a, &b, &c) == 3)
        -:   10:    {
        1:   11:        if (a != 0.0)
        -:   12:        {
        1:   13:            d = b * b - 4 * a * c;
        -:   14:
        1:   15:            if (d < 0.0)
        -:   16:            {
    #####:   17:                printf("There are no real roots\n");
        -:   18:            }
        1:   19:            else if (d > 0)
        -:   20:            {
    #####:   21:                printf("x1 = %f, x2 = %f\n", (-b - sqrt(d)) / (2 * a),
    #####:   22:                                             (-b + sqrt(d)) / (2 * a));
        -:   23:            }
        -:   24:            else
        -:   25:            {
        1:   26:                printf("x1 = x2 = %f\n", -b / (2 * a));
        -:   27:            }
        -:   28:        }
        -:   29:        else
        -:   30:        {
    #####:   31:            printf("Equation is not square\n");
        -:   32:        }
        -:   33:    }
        -:   34:    else
        -:   35:    {
    #####:   36:        printf("I/O error\n");
        -:   37:    }
        -:   38:
        1:   39:    return 0;
        -:   40:}
```