```c
#include <stdio.h>
int main(int argc, char *argv[])
{
  FILE *f;
  char  buf[256];

  if (argc != 2) {
    fprintf(stderr, "Wrong number of arguments\n");
    return 1;
  }
  if (!(f = fopen(argv[1], "r"))) {
    fprintf(stderr, "Cannot open file %s\n", argv[1]);
    return 1;
  }
  while (fgets(buf, sizeof(buf), f))
    fputs(buf, stdout);
  fclose(f);
  return 0;
}
```

Функция

```c
int ungetc(int c, FILE *stream);
```

Заталкивает один символ обратно во входной поток, так что следующая функция `getc` считает именно этот символ. Таким образом затолкнуть назад можно не более одного символа.

Функция

```c
int feof(FILE *stream);
```

возвращает ненулевое значение, если в структуре `FILE` установлен флаг ошибки или конца файла. Этот флаг устанавливается *по результату работы функций чтения или записи*. До первого чтения флаг сброшен, даже если файл пуст. В следующем примере последняя строка будет напечатана дважды.

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
        char  buf[64];
        while (!feof(stdin)) { // неверно!!!
                fgets(buf, sizeof(buf), stdin);
                fputs(buf, stdout);
        }
        return 0;
}
```

1.5 Упражнения

1. В аргументах командной строки задаются целые числа. Необходимо просуммировать их и напечатать результат на стандартный поток вывода.

2. В аргументах командной строки задаются имена текстовых файлов, содержащих последовательности вещественных файлов. Для каждого входного файла с именем *Name* создать выходной файл с именем *Name*`.max` и записать в него максимальное значение во входном файле.

1 Занятие №7

1.1 Вычисление выражений

Рассмотрим некоторые особенности вычисления выражений в языке **Си**.

1.1.1 Преобразования типов при вычислении выражений

Перед вычислением арифметических операций транслятор может выполнять два действия с аргументами: *целочисленное повышение* (integer promotion) и *балансировка*.

Рангом целого типа назовём число согласно таблице 1:

0	**_Bool**
1	**char**, **unsigned char**, **signed char**
2	**short**, **unsigned short**
3	**int**, **unsigned int**
4	**long**, **unsigned long**
5	**long long**, **unsigned long long**

Таблица 1: Ранги целых типов

Целочисленное повышение выполняется для аргументов типов **_Bool**, **cha** **unsigned char**, **signed char**, **short**, **unsigned short**, перечислимых типов, пре ставляемых указанными выше целочисленными типами, и битовых полей. Целочисленно повышение выполняется при передаче параметров, если соответствующий тип формально параметра не задан, а также для аргументов операций, за исключением операции **sizeof**.

Если тип аргумента, для которого выполняется целочисленное повышение, таков, что лю бое значение этого типа имеет представление в типе **int**, результат целочисленного пов шения имеет тип **int**, а в противном случае результат целочисленного повышения имеет т **unsigned int**. Практически это значит, что тип результата при выполнении целочисле ного повышения зависит от размеров целых типов в данном компиляторе языка Си. Напр мер, если размер типа **char** — 8 бит, **short** — 16 бит, **int** — 32 бита, то результат цел численного повышения всегда имеет тип **int**. Для архитектуры с размером типов **short** **int** 16 бит тип **short** повышается в тип **int**, а тип **unsigned short** — в тип **unsigne** **int**.

Балансировка. В случае вычисления инфиксного выражения, которое имеет два ариф метических операнда, транслятор определяет тип выражения с помощью балансировки т пов операндов. Для балансировки типов транслятор применяет следующие правила.

Для комплексного типа назовём *соответствующим вещественным типом* тип, базе которого построен данный комплексный тип. Наприме, для типа **float _Comple** соответствующим вещественным типом будет тип **float**. Для всех прочих типов *соответ ствующий вещественный тип* — это он сам.

- Если соответствующий вещественный тип одного из операндов — **long doubl** то соответствующий вещественный тип другого операнда преобразовывается в **lon** **double**. Если после этого один из операндов имеет тип **_Complex long doubl** то и другой операнд преобразовывается в тип **_Complex long double**. Таким о разом, например, сложение аргумента типов **_Complex float** и **long double** да результат типа **_Complex long double**.

- Иначе если соответствующий вещественный тип одного из операндов — **double**, то соответствующий вещественный тип другого операнда преобразовывается в **double**. Если после этого один из операндов имеет тип **_Complex double**, то и другой операнд преобразовывается в тип **_Complex double**.

- Иначе если соответствующий вещественный тип одного из операндов — **float**, то соответствующий вещественный тип другого операнда преобразовывается в **float**. Если после этого один из операндов имеет тип **_Complex float**, то и другой операнд преобразовывается в тип **_Complex float**.

- Иначе над обоими аргументами выполняется целочисленное повышение. Далее для определения типа аргументов и типа результата используются следующие правила.

  – Если типы аргументов совпадают, дальнейших преобразований не производится.

  – Иначе если оба типа аргументов — знаковые, или оба типа аргументов — беззнаковые, в качестве балансированного типа выбирается тот тип аргумента, ранг которого выше. Так, при сложении аргументов типа **int** и **long** результат будет иметь тип **long** независимо от размера каждого из типов.

  – Иначе если ранг типа аргумента, имеющего беззнаковый тип, не ниже ранга аргумента со знаковым типом, балансированный тип будет тип аргумента беззнакового типа. Например, при сложении аргументов типа **int** и **unsigned long** результат будет иметь тип **unsigned long**.

  – Иначе если тип операнда, имеющего знаковый тип, может представить все значения типа другого операнда, тогда балансированным типом будет знаковый тип. Например, на архитектуре с размером типа **long** 32 бита и размером типа **long long** 64 бита, результат сложения аргументов типа **unsigned long** и **long long** будет иметь тип **long long**.

  – Иначе балансированными типом будет беззнаковый тип, соответствующий знаковому типу. Например, на архитектуре с размером типов **int** и **long** 32 бита, результат сложения аргументов типа **unsigned int** и **long** будет иметь тип **unsigned long**.

Каждый из операндов преобразовывается в балансированный тип, арифметическая операция выполняется над значениями одинакового типа, и результат операции имеет балансированный тип.

Для вычислений с плавающей точкой может использоваться вещественный тип с большей точностью, чем требуется типом аргументов. Соответственно, промежуточные результаты могут храниться в этом типе. Это, однако, не изменяет балансированных типов аргументов операций.

Например, в предыдущих стандартах **Си** требовалось, чтобы вычисления с плавающей точкой велись в типе не менее точном, чем **double**. Сейчас это требование снято. С другой стороны, на архитектуре i386 вычисления с плавающей точкой часто ведутся в типе **long double**, так как в регистрах FPU значения хранятся в типе **long double**. Но если для вычислений с плавающей точкой используются инструкции SSE/SSE2, то используемый тип совпадает с типом выражение, то есть для сложения двух аргументов типа **float** используется инструкция сложения аргументов типа **float**.

открывает файл с именем `name`. Строка `mode` содержит флаги, с которыми открывается файл.

`"r"`	открыть для чтения. Текущая позиция в файле устанавливается на начало файла.
`"w"`	открыть для записи. Если файл не существовал, он создаётся, если существовал, он очищается. Текущая позиция в файле устанавливается на начало файла.
`"a"`	открыть для добавления. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла.
`"r+"`	открыть для чтения и записи. Текущая позиция в файле устанавливается на начало файла.
`"w+"`	открыть для чтения и записи. Если файл не существовал, он создаётся, если существовал, он очищается. Текущая позиция в файле устанавливается на начало файла.
`"a+"`	открыть для чтения и записи. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла.

Если файл был открыт успешно, возвращается указатель на структуру, хранящую состояние открытого файла. Если при открытии произошла ошибка, возвращается `NULL`.

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

Закрывает поток. Если закрытие прошло успешно, возвращается 0, иначе возвращается `EOF`.

При запуске программы открываются стандартные потоки `stdin`, `stdout`, `stderr`. Например, функции `putchar`, `puts` работают с потоком `stdin`, а функции `getchar`, `gets` работают с потоком `stdout`. Вы можете использовать эти имена для указания имён стандартных потоков в функциях, перечисленных ниже.

```
int getc(FILE *stream);
int putc(int c, FILE *stream);
int fscanf(FILE *stream, char *format, ...);
int fprintf(FILE *stream, char *format, ...);
```

Соответствуют функциям `getchar`, `putchar`, `scanf`, `printf`.

Функция

```
int fputs(char *str, FILE *stream);
```

записывает строку символов `str` в файл. Символ-терминатор строки отбрасывается. *В отличие от `puts` эта функция не дописывает `'\n'` в выходной файл.*

Функция

```
char *fgets(char *str, int size, FILE *stream);
```

Считывает самое большее `size - 1` символ из входного файла. Чтение останавливается по достижению конца файла или по достижению символа `'\n'`. Если `'\n'` считан, он добавляется в строку. После прочитанных символов добавляется символ-терминатор `'\0'`. При успешном завершении функция возвращает `str`, а при неудаче (конец файла или ошибка ввода) возвращается `NULL`. *Используйте эту функцию для чтения строк из входного файла, поскольку эта функция ограничивает максимальную длину считываемой строки.*

Следующий пример копирует содержимое заданного файла на стандартный вывод.

```
./prog1 f1 f2 ../f3
```

argc пример значение 4, argv[0] указывает на строку "./prog1", argv[1] — на строку "f1" и т. д.

Значение, возвращаемое функцией main (или аргумент функции exit), — это «код возврата» программы. Он используется чтобы проинформировать вызвавшую программу о статусе завершения работы программы. Код возврата 0 означает, что работа программы завершилась нормально, ненулевые коды возврата означают, что при выполнении программы возникли какие-то проблемы.

Пример программы, которая печатает свои аргументы.

```
1  #include <stdio.h>
2
3  int
4  main(int argc, char *argv[])
5  {
6          int i;
7
8          for (i = 0; i < argc; i++) {
9                  printf("argv[%d] = %s\n", i, argv[i]);
10         }
11         return 0;
12 }
```

1.4 Функции работы с файлами и потоками

Мы ещё не рассмотрели две функции для посимвольной работы со стандартным потоком. Функция

```
int getchar(void);
```

вводит один символ из стандартного входного потока. Если символ введён успешно, возвращается код символа в диапазоне 0–255 (положительное число). В случае ошибки или конца файла возвращается значение EOF (оно обычно равно -1). Функция

```
int putchar(int c);
```

выводит один символ на стандартный поток вывода.

Обратите внимание, что функция getchar может возвращать 257 различных значений, поэтому *переменной типа* ***char*** *для хранения возвращаемого значения недостаточно!* Следующий пример копирует стандартный ввод на стандартный вывод.

```
1  #include <stdio.h>
2  int main()
3  {
4          int c;
5          while ((c = getchar()) != EOF) putchar(c);
6          return 0;
7  }
```

Стандартная библиотека содержит средства для работы с произвольными файлами. Для хранения информации об открытом файле используется структура FILE. Функции работы с файлами принимают или возвращают указатель на эту структуру. Функция

```
FILE *fopen(char *name, char *mode);
```


Поскольку результат вычисления выражения с плавающей точкой зависит от поряд
выполнения операций и от типов промежуточных значений, на детали выполнения операц
с плавающей точкой необходимо обращать внимание.

1.1.2 Порядок вычисления выражения и побочные эффекты

Порядок, в котором транслятор вычисляет подвыражения неопределён и зависит от ре
лизации. Например, оператор

```
y = *p++;
```

может быть эквивалентно либо

```
temp = p; p += 1; y = *temp;
```

либо

```
y = *p; p += 1;
```

Если в программе встретилось выражение

```
f() + g()
```

компилятор может расположить вызовы функций f и g в произвольном порядке.

При вызове функции, например f(a, b), компилятор может вычислять выражения
произвольном порядке.

Порядок вычисления выражения важен, когда выражение имеет некоторый побочны
эффект, например, заносит значение в переменную, либо модифицирует состояние файла.

Программа на Си содержит *точки согласования*. В точках согласования точно извес
но, какие побочные эффекты имели место, а какие должны произойти. Например, кажд
выражение, записанное как оператор имеет точку согласования в конце. Гарантируется, ч
в фрагменте

```
y = 37;
x += y;
```

значение 37 будет помещено в y до того, как значение y будет использовано при вычи
лении x.

Точки согласования могут находится внутри выражения. Операции «запятая», выз
функции, логическое «и», логическое «или» содержат точки согласования. Например,

```
if ((c = getchar()) != EOF && isprint(c))
```

isprint(c) будет вычислено только после того, как новое значение, возвращённ
getchar(), будет занесено в переменную c.

Между двумя точками согласования один и тот же объект может модифицировать
только один раз, и значение, читаемое из модифицируемого объекта может использовать
ся только для вычисления нового значения этого объекта.

Например,

```
val = 10 * val + (c - '0'); // хорошо
i   = ++i + 2;              // плохо
```

1.2 Инициализация составных типов

Массивы, структуры и перечисления точно также, как и простые типы могут быть проинициализированы в точке определения. Массивы типов **char**, **signed char**, **unsigned char** могут быть проинициализированы строкой. Для одномерного массива произвольного типа инициализация выглядит следующим образом:

```
<тип> <имя>[<размер>] = { <зн. 1>, <зн. 2>, ... <зн. K> };
```

Все инициализирующие значения должны иметь тип, совпадающий с типом массива. Их не должно быть больше, чем размер массива, если их меньше, чем размер массива, оставшиеся элементы инициализируются по умолчанию. Например,

```
char str[20] = { 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0' };
```

Если массив содержит инициализацию, тогда размер массива может быть опущен. Он будет вычислен по количеству инициализирующих элементов.

Структура инициализируется похожим образом.

```
<тип> <имя> = { <зн. 1>, <зн. 2>, ..., <зн. K>};
```

В этом случае полям структуры последовательно присваиваются указанные значения. У объединений может быть проинициализирован только первый элемент.

Если массив сам имеет тип массива (многомерный массив), или тип структуры, или структура имеет поля типа массива или структуры, инициализаторы могут быть вложенными. Например

```
struct circle { double x, y, r; };
struct circle cc[2] = {{1.0, 2.0}, {1.3,2.0,0.4}};
```

Если фигурные скобки, отделяющие внутренние инициализаторы, опускаются, инициализация идёт подряд, переходя при необходимости границы типов. Например, если в указанном выше примере опустить внутренние фигурные скобки, `cc[1]` пример значение `{1.0,2.0,1.3}`, а `cc[2]` — `{2.0,0.4,0.0}` (если переменная `cc` — глобальная).

Обратите внимание, что если инициализирована только часть элементов локального массива, значение остальных элементов массива останется неопределённым. В следующем примере

```
void func(void)
{
  int x[10] = { 0, 0 };
}
```

значение элементов `x[0]` и `x[1]` будет установлено равным 0, а значение остальных элементов — неопределено!

Стандарт **C99** допускает инициализацию локальных переменных структурного и массивового типов, в которой инициализирующие выражения не являются константными. Например, следующий пример допустим в **C99**, но не допустим в **C90**.

```
void func(double x, double y)
{
  struct circle cc = { x, y, x + y };
}
```

Массивы переменного размера, однако, не могут быть проинициализированы.


```
void func(int n)
{
  int a[n] = { 1 }; // неверно!!!
}
```

В стандарте **C99** поддерживается явное указание имени поля или индекса массива при инициализации. Указание имени поля или индекса массива позволяет записывать инициализаторы в произвольном порядке. Например:

```
void func(int n)
{
  struct circle cc = { .r = 10, .x = 4 };
  int m[3] = { [1] = 1, [0] = 2 };
}
```

Неинициализированные элементы локальных объектов, как обычно, будут содержать неопределённые значения. Каждое поле или индекс массива может быть проинициализирован только раз. Если встречаются и инициализаторы с указанием индекса или поля и без них, индекс или имя поля вычисляется автоматически.

```
void func(int n)
{
  int m[3] = { [1] = 1, 2 };
}
```

В данном примере элемент `m[0]` будет не инициализирован, элемент `m[1]` получит значение `1`, а `m[2]` — `2`.

Другое новшество **C99** — возможность использования в выражениях *составных литералов* структурного и массивового типов. Составной литерал записывается следующим образом:

```
(<тип>) <инициализатор>
```

например

```
cc = (struct circle) { x + 1, y + 1, 5 };
```

Здесь выражения в инициализаторе также могут быть неконстантными.

Использование составного литерала эквивалентно созданию в текущей области видимости анонимной переменной соответствующего типа и её начальной инициализации указанным инициализатором. Анонимная переменная создаётся только один раз.

1.3 Передача параметров в программу

При запуске программы операционная система передаёт ей параметры, которые были указаны в командной строке. Если программа желает работать с переданными ей параметрами, заголовок функции `main` должен выглядеть следующим образом:

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

Здесь первый параметр `argc` задаёт количество аргументов командной строки, а параметр `argv` — это массив указателей на строки, хранящие значения соответствующих аргументов. По соглашению `argv[0]` содержит само имя программы. Следовательно, если программе не было передано никаких аргументов, `argv` равно 1. Разбиение командной строки на аргументы запускаемой программы производит командный процессор, обычно аргументы разделяются друг от друга пробелами. Например, если программа была запущена командо