Приложение B. Стандартная библиотека

B1. Ввод-вывод: <**stdio.h**>
B1.1. Операции над файлами
B1.2. Форматный вывод
B1.3. Форматный ввод
B1.4. Функции ввода-вывода символов
B1.5. Функции прямого ввода-вывода
B1.6. Функции позиционирования файла
B1.7. Функции обработки ошибок
B2. Проверки класса символа: <**ctype.h**>
B3. Функции, оперирующие со строками: <**string.h**>
B4. Математические функции: <**math.h**>
B5. Функции общего назначения: <**stdlib.h**>
B6. Диагностика: <**assert.h**>
B7. Списки аргументов переменной длины: <**stdarg.h**>
B8. Дальние переходы: <**setjmp.h**>
B9. Сигналы: <**signal.h**>
B10. Функции даты и времени: <**time.h**>
B11. Зависящие от реализации пределы: <**limits.h**> и <**float.h**>

Настоящее приложение представляет собой краткое изложение библиотеки, утвержденной в качестве ANSI-стандарта. Сама по себе библиотека не является частью языка, однако, заложенный в ней набор функций, а также определений типов и макросов составляет системную среду, поддерживающую стандарт Си. Мы не приводим здесь несколько функций с ограниченной областью применения – те, которые легко синтезируются из других функций, а также опускаем все то, что касается многобайтовых символов и специфики, обусловленной языком, национальными особенностями и культурой.

Функции, типы и макросы объявляются в следующих стандартных заголовочных файлах:

```
<assert.h>
<ctype.h>
<errno.h>

<float.h>
<limits.h>
<locale.h>

<math.h>
<setjmp.h>
<signal.h>

<stdarg.h>
<stddef.h>
<stdio.h>

<stdlib.h>
<string.h>
<time.h>
```

Доступ к заголовочному файлу осуществляется с помощью строки препроцессора

```
#include <заголовочный файл>
```

Заголовочные файлы можно включать в любом порядке и сколько угодно раз. Строка *#include* не должна быть внутри внешнего объявления или определения и должна встретиться раньше, чем что-нибудь из включаемого заголовочного файла будет востребовано. В конкретной реализации заголовочный файл может и не быть исходным файлом.

Внешние идентификаторы, начинающиеся со знака подчеркивания, а также все другие идентификаторы, начинающиеся с двух знаков подчеркивания или с подчеркивания и заглавной буквы, зарезервированы для использования в библиотеке.

B1. Ввод-вывод: <stdio.h>

Определенные в **<stdio.h>** функции ввода-вывода, а также типы и макросы составляют приблизительно одну треть библиотеки.

Поток - это источник или получатель данных; его можно связать с диском или с каким-то другим внешним устройством. Библиотека поддерживает два вида потоков: *текстовый* и *бинарный*, хотя на некоторых системах, в частности в UNIXe, они не различаются. *Текстовый поток* - это последовательность строк; каждая строка имеет нуль или более символов и заканчивается символом '\n'. Операционная среда может потребовать коррекции текстового потока (например, перевода '\n' в символы возврат-каретки и перевод-строки).

Бинарный поток - это последовательность непреобразованных байтов, представляющих собой некоторые промежуточные данные, которые обладают тем свойством, что если их записать, а затем прочесть той же системой ввода- вывода, то мы получим информацию, совпадающую с исходной.

Поток соединяется с файлом или устройством посредством его *открытия*, указанная связь разрывается путем *закрытия* потока. Открытие файла возвращает указатель на объект типа *FILE*, который содержит всю информацию, необходимую для управления этим потоком. Если не возникает двусмысленности, мы будем пользоваться терминами "файловый указатель" и "поток" как равнозначными.

Когда программа начинает работу, уже открыты три потока: **stdin**, **stdout** и **stderr**.

B1.1. Операции над файлами

Ниже перечислены функции, оперирующие с файлами. Тип **size_t** - беззнаковый целочисленный тип, используемый для описания результата оператора **sizeof**.

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

fopen открывает файл с заданным именем и возвращает поток или NULL, если попытка открытия оказалась неудачной. Режим *mode* допускает следующие значения:

"r"	- текстовый файл открывается для чтения (от *read* (англ.) - читать);
"w"	- текстовый файл создается для записи; старое содержимое (если оно было) выбрасывается (от *write* (англ.) - писать);
"a"	- текстовый файл открывается или создается для записи в конец файла (от *append* (англ.) - добавлять);
"r+"	- текстовый файл открывается для исправления (т. е. для чтения и для записи);
"w+"	- текстовый файл создается для исправления; старое содержимое (если оно было) выбрасывается;
"a+"	- текстовый файл открывается или создается для исправления уже существующей информации и добавления новой в конец файла.

Режим "исправления" позволяет читать и писать в один и тот же файл; при переходах от операций

чтения к операциям записи и обратно должны осуществляться обращения к **fflush** или к функции позиционирования файла. Если указатель режима дополнить буквой *b* (например "rb" или "w+b"), то это будет означать, что файл бинарный. Ограничение на длину имени файла задано константой FILENAME_MAX. Константа FOPEN_MAX ограничивает число одновременно открытых файлов.

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

freopen открывает файл с указанным режимом и связывает его с потоком *stream*. Она возвращает *stream* или, в случае ошибки, NULL. Обычно *freopen* используется для замены файлов, связанных с *stdin*, *stdout* или *stderr*, другими файлами.

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

Применяемая к потоку вывода функция **fflush** производит дозапись всех оставшихся в буфере (еще не записанных) данных, для потока ввода эта функция не определена. Возвращает EOF в случае возникшей при записи ошибки или нуль в противном случае. Обращение вида *fflush*(NULL) выполняет указанные операции для всех потоков вывода.

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

fclose производит дозапись еще не записанных буферизованных данных, сбрасывает несчитанный буферизованный ввод, освобождает все автоматически запрошенные буфера, после чего закрывает поток. Возвращает EOF в случае ошибки и нуль в противном случае.

```
int remove(const char *filename);
```

remove удаляет файл с указанным именем; последующая попытка открыть файл с этим именем вызовет ошибку. Возвращает ненулевое значение в случае неудачной попытки.

```
int rename(const char *oldname, const char *newname);
```

rename заменяет имя файла; возвращает ненулевое значение в случае, если попытка изменить имя оказалась неудачной. Первый параметр задает старое имя, второй - новое.

```
FILE *tmpfile(void);
```

tmpfile создает временный файл с режимом доступа "wb+", который автоматически удаляется при его закрытии или обычном завершении программой своей работы. Эта функция возвращает поток или, если не смогла создать файл, NULL.

```
char *tmpnam(char s[L_tmpnam]);
```

tmpnam(NULL) создает строку, не совпадающую ни с одним из имен существующих файлов, и возвращает указатель на внутренний статический массив. *tmpnam(s)* запоминает строку в *s* и возвращает ее в качестве значения функции; длина *s* должна быть не менее **L_tmpnam**. При каждом вызове *tmpnam* генерируется новое имя; при этом гарантируется не более TMPMAX различных имен за один сеанс работы программы. Заметим, что *tmpnam* создает имя, а не файл.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

setvbuf управляет буферизацией потока; к ней следует обращаться прежде, чем будет выполняться чтение, запись или какая-либо другая операция, *mode* со значением **_IOFBF** вызывает полную буферизацию, с **_IOLBF** - "построчную" буферизацию текстового файла, а *mode* со значением **_IONBF** отменяет всякую буферизацию. Если параметр *buf* не есть NULL, то его значение - указатель на буфер, в противном случае под буфер будет запрашиваться память. Параметр *size* задает размер буфера. Функция *setvbuf* в случае ошибки выдает ненулевое значение.

```
void setbuf(FILE *stream, char *buf);
```

Если *buf* есть NULL, то для потока *stream* буферизация выключается. В противном случае вызов **setbuf** приведет к тем же действиям, что и вызов (void) setvbuf (stream, buf, _IOFBF, BUFSIZ).

B1.2. Форматный вывод

Функции **printf** осуществляют вывод информации по формату.

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

fprintf преобразует и пишет вывод в поток *stream* под управлением *format*. Возвращаемое значение - число записанных символов или, в случае ошибки, отрицательное значение.

Форматная строка содержит два вида объектов: *обычные символы*, копируемые в выводной поток, и *спецификации преобразования*, которые вызывают преобразование и печать остальных аргументов в том порядке, как они перечислены. Каждая спецификация преобразования начинается с **%** и заканчивается символом-спецификатором преобразования. Между % и символом- спецификатором в порядке, в котором они здесь перечислены, могут быть расположены следующие элементы информации:

- Флаги (в любом порядке), модифицирующие спецификацию:

-	- указывает на то, что преобразованный аргумент должен быть прижат к левому краю поля;
+	- предписывает печатать число всегда со знаком;
пробел	- если первый символ - не знак, то числу должен предшествовать пробел;
0	- указывает, что числа должны дополняться слева нулями до всей ширины поля;
#	- указывает на одну из следующих форм вывода: для *o* первой цифрой должен быть 0; для *x* или *X* ненулевому результату должны предшествовать 0x или 0X; для *e*, *E*, *f*, *g* и *G* вывод должен обязательно содержать десятичную точку; для *g* и *G* завершающие нули не отбрасываются.

- Число, специфицирующее минимальную ширину поля. Преобразованный аргумент будет напечатан в поле, размер которого не меньше указанной ширины, а если потребуется, в поле большего размера. Если число символов преобразованного аргумента меньше ширины поля, то поле будет дополнено слева (или справа, если число прижимается к левому краю). Обычно поле дополняется пробелами (или нулями, если присутствует флаг дополнения нулями).
- Точка, отделяющая указатель ширины поля от указателя точности.
- Число, задающее точность, которое специфицирует максимальное количество символов, печатаемых из строки, или количество цифр после десятичной точки в преобразованиях e, E или f, или количество значащих цифр для g или G - преобразования, или минимальное количество цифр при печати целого (до необходимой ширины поля число дополняется слева нулями).
- Модификаторы **h**, **l** (буква ell) или **L**. "h" указывает на то, что соответствующий аргумент должен печататься как *short* или *unsigned short*; "l" сообщает, что аргумент имеет тип *long* или *unsigned long*; "L" информирует, что аргумент принадлежит типу *long double*.

Ширина, или точность, или обе эти характеристики могут быть специфицированы с помощью *****; в этом случае необходимое число "извлекается" из следующего аргумента, который должен иметь тип *int* (в случае двух звездочек используются два аргумента).

Символы-спецификаторы и разъяснение их смысла приведены в таблице B-1. Если за **%** нет правильного символа-спецификатора, результат не определен.

```
int printf(const char *format, ...);
```

printf(...) полностью эквивалентна *fprintf*(stdout, ...).

```
int sprintf(char *s, const char *format, ...)
```

sprintf действует так же, как и *printf*, только вывод осуществляет в строку *s*, завершая ее символом '\0'. Строка *s* должна быть достаточно большой, чтобы вмещать результат вывода. Возвращает количество записанных символов, в число которых символ '\0' не входит.

```
int vprintf (const char *format, va_list arg)
int vfprintf (FILE *stream, const char *format, va_list arg)
int vsprintf (char *s, const char *format, va_list arg)
```

Функции **vprintf**, **vfprintf** и **vsprintf** эквивалентны соответствующим *printf*- функциям с той лишь

разницей, что переменный список аргументов представлен параметром *arg*, инициализированным макросом *va_start* и, возможно, вызовами *va_arg* (см. в B7 описание <**stdarg.h**>).

Таблица B-1. Преобразования *printf*

Символ	Тип аргумента; вид печати
d, **i**	**int**; знаковая десятичная запись
o	**unsigned int**; беззнаковая восьмеричная запись (без 0 слева)
x, **X**	**unsigned int**; беззнаковая шестнадцатеричная запись (без 0x или 0X слева), в качестве цифр от 10 до 15 используются abcdef для x и ABCDEF для X
u	**unsigned int**; беззнаковое десятичное целое
c	**int**; единичный символ после преобразования в *unsigned char*
s	**char ***; символы строки печатаются, пока не встретится '\0' или не исчерпается количество символов, указанное точностью
f	**double**; десятичная запись вида [-]mmm.ddd, где количество *d* специфицируется точностью. По умолчанию точность равна 6; нулевая точность подавляет печать десятичной точки
e, **E**	**double**; десятичная запись вида [-]m.dddddde±xx или запись вида [-]m.ddddddE±xx, где количество *d* специфицируется точностью. По умолчанию точность равна 6; нулевая точность подавляет печать десятичной точки
g, **G**	**double**; используется **%e** и **%E**, если порядок меньше -4 или больше или равен точности; в противном случае используется **%f**. Завершающие нули и точка в конце не печатаются
p	**void ***; печатает в виде указателя (представление зависит от реализации
n	**int ***; число символов, напечатанных к данному моменту данным вызовом *printf*, *записывается* в аргумент. Никакие другие аргументы не преобразуются
%	никакие аргументы не преобразуются; печатается **%**

B1.3. Форматный ввод

Функции **scanf** имеют дело с форматным преобразованием при вводе

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

fscanf читает данные из потока *stream* под управлением *format* и преобразованные величины присваивает по порядку аргументам, каждый из которых должен быть указателем. Завершает работу, если исчерпался формат. Выдает EOF по исчерпании файла или перед любым преобразованием, если возникла ошибка; в остальных случаях функция возвращает количество преобразованных и введенных элементов.

Форматная строка обычно содержит спецификации преобразования, которые используются для управления вводом. В форматную строку могут входить:

- пробелы и табуляции, которые игнорируются;
- обычные символы (кроме **%**), которые ожидаются в потоке ввода среди символов, отличных от символов-разделителей;
- спецификации преобразования, состоящие из **%**; необязательного знака *****, подавляющего присваивание; необязательного числа, специфицирующего максимальную ширину поля; необязательных **h**, **l** или **L**, указывающих размер присваиваемого значения, и символа-спецификатора преобразования.

Спецификация преобразования определяет преобразование следующего поля ввода. Обычно результат размещается в переменной, на которую указывает соответствующий аргумент. Однако если

присваивание подавляется с помощью знака *, как, например, в %*s, то поле ввода просто пропускается, и никакого присваивания не происходят. Поле ввода определяется как строка символов, отличных от символов-разделителей; при этом ввод строки прекращается при выполнении любого из двух условий: если встретился символ-разделитель или если ширина поля (в случае, когда она указана) исчерпана. Из этого следует, что при переходе к следующему полю *scanf* может "перешагивать" через границы строк, поскольку символ новой строки является символом-разделителем. (Под символами-разделителями понимаются символы пробела, табуляции, новой строки, возврата каретки, вертикальной табуляции и смены страницы.)

Символ-спецификатор указывает на способ интерпретации поля ввода. Соответствующий аргумент должен быть указателем. Список допустимых символов-спецификаторов приводится в таблице B-2.

Символам-спецификаторам **d**, **i**, **n**, **o**, **u** и **x** может предшествовать **h**, если аргумент есть указатель на *short* (а не *int*) или **l** (буква ell), если аргумент есть указатель на *long*. Символам-спецификаторам **e**, **f** и **g** может предшествовать **l**, если аргумент - указатель на *double* (а не *float*), или **L**, если аргумент - указатель на *long double*.

```
int scanf (const char *format, ...);
```

scanf(...) делает то же, что и *fscanf*(stdin, ...).

```
int sscanf (const char *s, const char *format, ...);
```

sscanf(s, ...) делает то же, что и *scanf*(...), только ввод символов осуществляет из строки *s*.

Таблица B-2. Преобразования *scanf*

Символ	Данные на вводе; тип аргумента
d	десятичное целое; **int ***
i	целое: **int ***. Целое может быть восьмеричным (с нулем слева) или шестнадцатеричным (с 0x или 0X слева)
o	восьмеричное целое (с нулем слева или без него); **int ***
u	беззнаковое десятичное целое; **unsigned int ***
x	шестнадцатеричное целое (с 0x или 0X слева или без них): **int ***
c	символы, **char ***. Символы ввода размещаются в указанном массиве в количестве, заданном шириной поля; по умолчанию это количество равно 1. Символ'\0' не добавляется. Символы-разделители здесь рассматриваются как обычные символы и поступают в аргумент. Чтобы прочесть следующий символ-разделитель, используйте %1s
s	строка символов, отличных от символов-разделителей (записывается без кавычек); **char ***, указывающий на массив размера достаточного, чтобы вместить строку и добавляемый к ней символ '\0'
e, **f**, **g**	число с плавающей точкой; **float ***. Формат ввода для *float* состоит из необязательного знака, строки цифр, возможно с десятичной точкой, и необязательного порядка, состоящего из **E** или **e** и целого, возможно со знаком
p	значение указателя в виде, в котором printf ("%p") его напечатает; **void ***
n	записывает в аргумент число символов, прочитанных к этому моменту в этом вызове; **int ***. Никакого чтения ввода не происходит. Счетчик числа введенных элементов не увеличивается
[...]	выбирает из ввода самую длинную непустую строку, состоящую из символов, заданных в квадратных скобках: **char ***. В конец строки добавляется '\0'. Спецификатор вида **[]...]** включает **]** в задаваемое множество символов

[^...]	выбирает из ввода самую длинную непустую строку, состоящую из символов, не входящих в заданное в скобках множество. В конец добавляется '\0'. Спецификатор вида **[^]...]** включает **]** в задаваемое множество символов
%	обычный символ **%**; присваивание не делается

B1.4. Функции ввода-вывода символов

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

fgetc возвращает следующий символ из потока *stream* в виде *unsigned char* (переведенную в *int*) или EOF, если исчерпан файл или обнаружена ошибка.

```
char *fgets(char *s, int n, FILE *stream);
```

fgets читает не более *n-1* символов в массив *s*, прекращая чтение, если встретился символ новой строки, который включается в массив; кроме того, записывает в массив '\0'. Функция *fgets* возвращает *s* или, если исчерпан файл или обнаружена ошибка, NULL.

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

fputc пишет символ *c* (переведенный в *unsigned char*) в *stream*. Возвращает записанный символ или EOF в случае ошибки.

```
int fputs(const char *s, FILE *stream);
```

fputs пишет строку *s* (которая может не иметь '\n') в *stream*; возвращает неотрицательное целое или EOF в случае ошибки.

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

getc делает то же, что и *fgetc*, но в отличие от последней, если она - макрос, *stream* может браться более одного раза.

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

getchar() делает то же, что *getc*(stdin).

```
char *gets(char *s);
```

gets читает следующую строку ввода в массив *s*, заменяя символ новой строки на '\0'. Возвращает *s* или, если исчерпан файл или обнаружена ошибка, NULL.

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

putc делает то же, что и *fputc*, но в отличие от последней, если *putc* - макрос, значение *stream* может браться более одного раза.

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

putchar(c) делает тоже, что *putc*(c, stdout).

```
int puts(const char *s);
```

puts пишет строку *s* и символ новой строки в *stdout*. Возвращает EOF в случае ошибки, или неотрицательное значение, если запись прошла нормально.

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

ungetc отправляет символ *c* (переведенный в *unsigned char*) обратно в *stream*; при следующем чтении из *stream* он будет получен снова. Для каждого потока вернуть можно не более одного символа. Нельзя возвращать EOF. В качестве результата *ungetc* выдает отправленный назад символ или, в случае ошибки, EOF.

B1.5. Функции прямого ввода-вывода

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream);
```

fread читает из потока *stream* в массив *ptr* не более *nobj* объектов размера *size*. Она возвращает количество прочитанных объектов, которое может быть меньше заявленного. Для индикации состояния после чтения следует использовать *feof* и *ferror*.

```
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream);
```

fwrite пишет из массива *ptr* в *stream nobj* объектов размера *size*; возвращает число записанных объектов, которое в случае ошибки меньше *nobj*.

B1.6. Функции позиционирования файла

```
int fseek(FILE *stream, long offset, int origin);
```

fseek устанавливает позицию для *stream*; последующее чтение или запись будет производиться с этой позиции. В случае бинарного файла позиция устанавливается со смещением *offset* - относительно начала, если *origin* равен **SEEK_SET**; относительно текущей позиции, если *origin* равен **SEEK_CUR**; и относительно конца файла, если *origin* равен **SEEK_END**. Для текстового файла *offset* должен быть нулем или значением, полученным с помощью вызова функции *ftell*. При работе с текстовым файлом *origin* всегда должен быть равен **SEEK_SET**.

```
long ftell(FILE *stream);
```

ftell возвращает текущую позицию потока *stream* или -1L, в случае ошибки.

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

rewind(fp) делает то же, что и fseek(fp, 0L, SEEK_SET); clearerr(fp).

```
int fgetpos(FILE *stream, fpos_t *ptr);
```

fgetpos записывает текущую позицию потока *stream* в **ptr* для последующего использования ее в *fsetpos*. Тип **fpos_t** позволяет хранить такого рода значения, В случае ошибки *fgetpos* возвращает ненулевое значение.

```
int fsetpos(FILE *stream, const fpos_t *ptr);
```

fsetpos устанавливает позицию в *stream*, читая ее из **ptr*, куда она была записана ранее с помощью *fgetpos*. В случае ошибки *fsetpos* возвращает ненулевое значение.

B1.7. Функции обработки ошибок

Многие функции библиотеки в случае ошибки или конца файла устанавливают индикаторы состояния. Эти индикаторы можно проверять и изменять. Кроме того, целое выражение **errno** (объявленное в <**errno.h**>) может содержать номер ошибки, который дает дополнительную информацию о последней из обнаруженных ошибок.

```
void clearerr(FILE *stream);
```

clearerr очищает индикаторы конца файла и ошибки потока *stream*.

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

feof возвращает ненулевое значение, если для потока *stream* установлен индикатор конца файла.

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

ferror возвращает ненулевое значение, если для потока *stream* установлен индикатор ошибки.

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

perror(s) печатает *s* и зависимое от реализации сообщение об ошибке, соответствующее целому значению в *errno*, т. е. делает то же, что и обращение к функции *fprintf* вида

```
fprintf(stderr, "%s: %s\n", s, "сообщение об ошибке")
```

См. *strerror* в параграфе B3.

B2. Проверки класса символа: <ctype.h>

Заголовочный файл **<ctype.h>** объявляет функции, предназначенные для проверок символов. Аргумент каждой из них имеет тип *int* и должен либо представлять собой EOF, либо быть значением *unsigned char*, приведенным к *int*; возвращаемое значение тоже имеет тип *int*. Функции возвращают ненулевое значение ("истина"), когда аргумент с удовлетворяет описанному условию или принадлежит указанному классу символов, и нуль в противном случае.

isalnum(c)	**isalpha**(c) или **isdigit**(c) есть истина
isalpha(c)	**isupper**(c) или **islower**(c) есть истина
iscntrl(c)	управляющий символ
isdigit(c)	десятичная цифра
isgraph(c)	печатаемый символ кроме пробела
islower(c)	буква нижнего регистра
isprint(c)	печатаемый символ, включая пробел
ispunct(c)	печатаемый символ кроме пробела, буквы или цифры
isspace(c)	пробел, смена страницы, новая строка, возврат каретки, табуляция, вертикальная табуляция
isupper(c)	буква верхнего регистра
isxdigit(c)	шестнадцатеричная цифра

В наборе семибитовых ASCII-символов печатаемые символы находятся в диапазоне от **0x20** (' ') до **0x7E** ('~'); управляющие символы - от **0** (*NUL*) до **0x1F** (*US*) и 0x7F (*DEL*).

Помимо перечисленных есть две функции, приводящие буквы к одному из регистров:

```
int tolower(int c) – переводит c на нижний регистр;
int toupper(int c) - переводит c на верхний регистр.
```

Если *c* - буква на верхнем регистре, то *tolower*(c) выдаст эту букву на нижнем регистре; в противном случае она вернет *c*. Если *c* - буква на нижнем регистре, то *toupper*(c) выдаст эту букву на верхнем регистре; в противном случае она вернет *c*.

B3. Функции, оперирующие со строками: <string.h>

Имеются две группы функций, оперирующих со строками. Они определены в заголовочном файле **<string.h>**. Имена функций первой группы начинаются с букв *str*, второй - с *mem*. Если копирование имеет дело с объектами, перекрывающимися по памяти, то, за исключением **memmove**, поведение функций не определено. Функции сравнения рассматривают аргументы как массивы элементов типа *unsigned char*.

В таблице на с. 321 переменные *s* и *t* принадлежат типу *char **, *cs* и *ct* – типу *const char **, *n* - типу *size_t*, а *c* - значение типа *int*, приведенное к типу *char*.

Последовательные вызовы **strtok** разбивают строку *s* на лексемы. Ограничителем лексемы служит любой символ из строки *ct*. В первом вызове указатель *s* не равен NULL. Функция находит в строке *s* первую лексему, состоящую из символов, не входящих в *ct*; ее работа заканчивается тем, что поверх следующего символа пишется '\0' и возвращается указатель на лексему. Каждый последующий вызов, в

котором указатель *s* равен NULL, возвращает указатель на следующую лексему, которую функция будет искать сразу за концом предыдущей. Функция *strtok* возвращает NULL, если далее никакой лексемы не обнаружено. Параметр *ct* от вызова к вызову может варьироваться.

Здесь и ниже под такими выражениями как cs<ct не следует понимать арифметическое сравнение указателей. Подразумевается лексикографическое сравнение, т. е. *cs* меньше (больше) *ct*, если первый несовпавший элемент в *cs* арифметически меньше (больше) соответствующего элемента из *ct*.— *Примеч. ред.*

char ***strcpy**(s,ct)	копирует строку *ct* в строку *s*, включая '\0'; возвращает *s*
char ***strncpy**(s,ct,n)	копирует не более *n* символов строки *ct* в *s*; возвращает *s*. Дополняет результат символами '\0', если символов в *ct* меньше *n*
char ***strcat**(s,ct)	приписывает *ct* к *s*; возвращает *s*
char ***strncat**(s,ct,n)	приписывает не более *n* символов *ct* к *s*, завершая *s* символом '\0'; возвращает *s*
char **strcmp**(cs,st)	сравнивает *cs* и *ct*; возвращает <0, если cs<ct; 0, если cs==ct; и >0, если cs>ct (*I.B.: вообще-то, функция возвращает int*)
char **strncmp**(cs,ct)	сравнивает не более *n* символов *cs* и *ct*; возвращает <0, если cs<ct, 0, если cs==ct, и >0, если cs>ct (*I.B.: тоже int должна возвращать*)
char ***strchr**(cs,c)	возвращает указатель на первое вхождение *c* в *cs* или, если такового не оказалось, NULL
char ***strrchr**(cs,c)	возвращает указатель на последнее вхождение *c* в *cs* или, если такового не оказалось, NULL
size_t **strspn**(cs,ct)	возвращает длину начального сегмента *cs*, состоящего из символов, входящих в строку *ct*
size_t **strcspn**(cs,ct)	возвращает длину начального сегмента *cs*, состоящего из символов, не входящих в строку *ct*
char ***strpbrk**(cs,ct)	возвращает указатель в *cs* на первый символ, который совпал с одним из символов, входящих в *ct*, или, если такового не оказалось, NULL
char ***strstr**(cs, ct)	возвращает указатель на первое вхождение *ct* в *cs* или, если такового не оказалось, NULL
size_t **strlen**(cs)	возвращает длину *cs*
char * **strerror**(n)	возвращает указатель на зависящую от реализации строку, соответствующую номеру ошибки *n*
char * **strtok**(s, ct)	*strtok* ищет в *s* лексему, ограниченную символами из *ct*; более подробное описание этой функции см. ниже

Функции **mem...** предназначены для манипулирования с объектами как с массивами символов; их назначение - получить интерфейсы к эффективным программам. В приведенной ниже таблице *s* и *t* принадлежат типу *void **; *cs* и *ct* - типу *const void **; *n* - типу *size_t*; а *c* имеет значение типа *int*, приведенное к типу *char*.

void ***memcpy**(s,ct, n)	копирует *n* символов из *ct* в *s* и возвращает *s*
void ***memmove**(s,ct,n)	делает то же самое, что и *memcpy*, но работает и в случае "перекрывающихся" объектов.
int **memcmp**(cs, ct, n)	сравнивает первые *n* символов *cs* и *ct*; выдает тот же результат, что и функция *strcmp*

void ***memchr**(cs, c, n)	возвращает указатель на первое вхождение символа *c* в *cs* или, если среди первых *n* символов *c* не встретилось, NULL
void ***memset**(s, c, n)	размещает символ *c* в первых *n* позициях строки *s* и возвращает *s*

B4. Математические функции: <math.h>

В заголовочном файле **<math.h>** описываются математические функции и определяются макросы.

Макросы **EDOM** и **ERANGE** (находящиеся в **<errno.h>**) задают отличные от нуля целочисленные константы, используемые для фиксации ошибки области и ошибки диапазона; **HUGE_VAL** определена как положительное значение типа **double**. *Ошибка области* возникает, если аргумент выходит за область значений, для которой определена функция. Фиксация ошибки области осуществляется присвоением *errno* значения *EDOM*; возвращаемое значение зависит от реализации. Ошибка диапазона возникает тогда, когда результат функции не может быть представлен в виде *double*. В случае переполнения функция возвращает *HUGE_VAL* с правильным знаком и в *errno* устанавливается значение *ERANGE*. Если результат оказывается меньше, чем возможно представить данным типом, функция возвращает нуль, а устанавливается ли в этом случае *errno* в *ERANGE*, зависит от реализации. Далее *x* и *y* имеют тип *double*, *n* - тип *int*, и все функции возвращают значения типа *double*. Углы в тригонометрических функциях задаются в *радианах*.

sin(x)	синус *x*
cos(x)	косинус *x*
tan(x)	тангенс *x*
asin(x)	арксинус *x* в диапазоне *[-pi/2,pi/2]*, *x* в диапазоне *[-1,1]*
acos(x)	арккосинус *x* в диапазоне *[0, pi]*, *x* в диапазоне *[-1,1]*
atan(x)	арктангенс *x* в диапазоне *[-pi/2, pi/2]*
atan2(y,x)	арктангенс *y/x* в диапазоне *[-pi, pi]*
sinh(x)	гиперболический синус *x*
cosh(x)	гиперболический косинус *x*
tanh(x)	гиперболический тангенс *x*
exp(x)	Экспоненциальная функция e^x
log(x)	натуральный логарифм *ln(x), x > 0*
log10(x)	десятичный логарифм *lg(x), x > 0*
pow(x,y)	x^y, ошибка области, если *x = 0* или *y<=0* или *x<0* и *y* – не целое
sqrt(x)	квадратный корень *x*, *x >= 0*
ceil(x)	наименьшее целое в виде *double*, которое не меньше *x*
floor(x)	наибольшее целое в виде *double*, которое не больше *x*
fabs(x)	абсолютное значение |x|
ldexp(x, n)	$x * 2^n$
frexp(x, int *exp)	разбивает *x* на два сомножителя, первый из которых - нормализованная дробь в интервале [1/2, 1), которая возвращается, а второй - степень двойки, эта степень запоминается в **exp*. Если *x* - нуль, то обе части результата равны нулю

modf(x,double *ip)	разбивается на целую и дробную части, обе имеют тот же знак, что и *x*. Целая часть запоминается в **ip*, дробная часть выдается как результат
fmod(x, y)	остаток от деления *x* на *y* в виде числа с плавающей точкой. Знак результата совпадает со знаком *x*. Если *y* равен нулю, результат зависит от реализации

B5. Функции общего назначения: <stdlib.h>

Заголовочный файл **<stdlib.h>** объявляет функции, предназначенные для преобразования чисел, запроса памяти и других задач.

```
double atof(const char *s)
```

atof переводит *s* в *double*; эквивалентна strtod(s, (char**) NULL).

```
int atoi(const char *s)
```

atoi переводит *s* в *int*; эквивалентна (int)strtol(s, (char**)NULL, 10).

```
int atol(const char *s)
```

atol переводит *s* в *long*; эквивалентна strtol(s, (char**) NULL, 10).

```
double strtod(const char *s, char **endp)
```

strtod преобразует первые символы строки *s* в *double*, игнорируя начальные символы-разделители; запоминает указатель на непреобразованный конец в **endp* (если *endp* не NULL), при переполнении она выдает *HUGE_VAL* с соответствующим знаком, в случае, если результат оказывается меньше, чем возможно представить данным типом, возвращается 0; в обоих случаях в *errno* устанавливается *ERANGE*.

```
long strtol(const char *s, char **endp, int base)
```

strtol преобразует первые символы строки *s* в *long*, игнорируя начальные символы-разделители; запоминает указатель на непреобразованный конец в **endp* (если *endp* не NULL). Если *base* находится в диапазоне от 2 до 36, то преобразование делается в предположении, что на входе - запись числа по основанию *base*. Если *base* равно нулю, то основанием числа считается 8, 10 или 16; число, начинающееся с цифры 0, считается восьмеричным, а с 0x или 0X - шестнадцатеричным. Цифры от 10 до *base-1* записываются начальными буквами латинского алфавита в любом регистре. При основании, равном 16, в начале числа разрешается помещать 0x или 0X. В случае переполнения функция возвращает LONG_MAX или LONG_MIN (в зависимости от знака), а в *errno* устанавливается ERANGE.

```
unsigned long strtoul(const char *s, char **endp, int base)
```

strtoul работает так же, как и *strtol*, с той лишь разницей, что возвращает результат типа *unsigned long*, а в случае переполнения - ULONG_MAX.

```
int rand(void)
```

rand выдает псевдослучайное число в диапазоне от 0 до RAND_MAX; RAND_MAX не меньше 32767.

```
void srand(unsigned int seed)
```

srand использует *seed* в качестве семени для новой последовательности псевдослучайных чисел. Изначально параметр *seed* равен 1.

```
void *calloc(size_t nobj, size_t size)
```

calloc возвращает указатель на место в памяти, отведенное для массива *nobj* объектов, каждый из которых размера *size*, или, если памяти запрашиваемого объема нет, NULL. Выделенная область памяти обнуляется.

```
void *malloc(size_t size)
```

malloc возвращает указатель на место в памяти для объекта размера *size* или, если памяти

запрашиваемого объема нет, NULL. Выделенная область памяти не инициализируется.

```
void *realloc(void *p, size_t size)
```

realloc заменяет на *size* размер объекта, на который указывает *p*. Для части, размер которой равен наименьшему из старого и нового размеров, содержимое не изменяется. Если новый размер больше старого, дополнительное пространство не инициализируется, *realloc* возвращает указатель на новое место памяти или, если требования не могут быть удовлетворены, NULL (**p* при этом не изменяется).

```
void free(void *p)
```

free освобождает область памяти, на которую указывает *p*; эта функция ничего не делает, если *p* равно NULL. В *p* должен стоять указатель на область памяти, ранее выделенную одной из функций: *calloc*, *malloc* или *realloc*.

```
void abort(void *p)
```

abort вызывает аварийное завершение программы, ее действия эквивалентны вызову **raise**(SIGABRT).

```
void exit(int status)
```

exit вызывает нормальное завершение программы. Функции, зарегистрированные с помощью **atexit**, выполняются в порядке, обратном их регистрации. Производится опорожнение буферов открытых файлов, открытые потоки закрываются, и управление возвращается в среду, из которой был произведен запуск программы. Значение *status*, передаваемое в среду, зависит от реализации, однако при успешном завершении программы принято передавать нуль. Можно также использовать значения EXIT_SUCCESS (в случае успешного завершения) и EXIT_FAILURE (в случае ошибки).

```
int atexit(void (*fcn)(void))
```

atexit регистрирует *fcn* в качестве функции, которая будет вызываться при нормальном завершении программы; возвращает ненулевое значение, если регистрация не может быть выполнена.

```
int system(const char *s)
```

system передает строку *s* операционной среде для выполнения. Если *s* есть NULL и существует командный процессор, то *system* возвращает ненулевое значение. Если *s* не NULL, то возвращаемое значение зависит от реализации.

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

getenv возвращает строку среды, связанную с *name*, или, если никакой строки не существует, NULL. Детали зависят от реализации.

```
void *bsearch(const void *key, const void *base,
    size_t n, size_t size,
    int (*cmp)(const void *keyval, const void *datum))
```

bsearch ищет среди *base[0]...base[n-1]* элемент с подходящим ключом **key*. Функция *cmp* должна сравнивать первый аргумент (ключ поиска) со своим вторым аргументом (значением ключа в таблице) и в зависимости от результата сравнения выдавать отрицательное число, нуль или положительное значение. Элементы массива *base* должны быть упорядочены в возрастающем порядке, *bsearch* возвращает указатель на элемент с подходящим ключом или, если такого не оказалось, NULL.

```
void qsort(void *base, size_t n, size_t size,
    int (*cmp)(const void *, const void *))
```

qsort сортирует массив *base[0]...base[n-1]* объектов размера *size* в возрастающем порядке. Функция сравнения *cmp* - такая же, как и в *bsearch*.

```
int abs(int n)
```

abs возвращает абсолютное значение аргумента типа *int*.

```
long labs(long n)
```

labs возвращает абсолютное значение аргумента типа *long*.

```
div_t div(int num, int denom)
```

div вычисляет частное и остаток от деления *num* на *denom*. Результаты типа *int* запоминаются в элементах *quot* и *rem* структуры *div_t*.

```
ldiv_t ldiv(long num, long denom)
```

ldiv вычисляет частное и остаток от деления *num* на *denom*. Результаты типа *long* запоминаются в элементах *quot* и *rem* структуры *ldiv_t*.

B6. Диагностика: <assert.h>

Макрос **assert** используется для включения в программу диагностических сообщений.

```
void assert (int выражение)
```

Если *выражение* имеет значение нуль, то

```
assert (выражение)
```

напечатает в *stderr* сообщение следующего вида:

```
Assertion failed: выражение, file имя-файла, line nnn
```

после чего будет вызвана функция *abort*, которая завершит вычисления. Имя исходного файла и номер строки будут взяты из макросов __FILE__ и __LINE__.

Если в момент включения файла <**assert.h**> было определено имя **NDEBUG**, то макрос *assert* игнорируется.

B7. Списки аргументов переменной длины: <stdarg.h>

Заголовочный файл <**stdarg.h**> предоставляет средства для перебора аргументов функции, количество и типы которых заранее не известны. Пусть *lastarg* - последний именованный параметр функции *f* с переменным числом аргументов. Внутри *f* объявляется переменная *ap* типа **va_list**, предназначенная для хранения указателя на очередной аргумент:

```
va_list ap;
```

Прежде чем будет возможен доступ к безымянным аргументам, необходимо один раз инициализировать *ap*, обратившись к макросу **va_start**:

```
va_start(va_list ap, lastarg);
```

С этого момента каждое обращение к макросу:

```
type va_arg(va_list ap, type);
```

будет давать значение очередного безымянного аргумента указанного типа, и каждое такое обращение будет вызывать автоматическое приращение указателя *ap*, чтобы последний указывал на следующий аргумент. Один раз после перебора аргументов, но до выхода из *f* необходимо обратиться к макросу

```
void va_end(va_list ap);
```

B8. Дальние переходы: <setjmp.h>

Объявления в <**setjmp.h**> предоставляют способ отклониться от обычной последовательности "вызов - возврат"; типичная ситуация - необходимость вернуться из "глубоко вложенного" вызова функции на верхний уровень, минуя промежуточные возвраты.

```
int setjmp(jmp_buf env);
```

Макрос **setjmp** сохраняет текущую информацию о вызовах в *env* для последующего ее использования в **longjmp**. Возвращает нуль, если возврат осуществляется непосредственно из *setjmp*, и не нуль, если - от

последующего вызова *longjmp*. Обращение к *setjmp* возможно только в определенных контекстах, в основном это проверки в *if*, *switch* и циклах, причем только в простых выражениях отношения.

```
if (setjmp() == 0)
    /* после прямого возврата */
else
    /* после возврата из longjmp */


void longjmp(jmp_buf env, int val);
```

longjmp восстанавливает информацию, сохраненную в самом последнем вызове *setjmp*, по информации из *env*; выполнение программы возобновляется, как если бы функция *setjmp* только что отработала и вернула ненулевое значение *val*. Результат будет непредсказуемым, если в момент обращения к *longjmp* функция, содержащая вызов *setjmp*, уже "отработала" и осуществила возврат. Доступные ей объекты имеют те значения, которые они имели в момент обращения к *longjmp*; *setjmp* не сохраняет значений.

B9. Сигналы: <signal.h>

Заголовочный файл <**signal.h**> предоставляет средства для обработки исключительных ситуаций, возникающих во время выполнения программы, таких как прерывание, вызванное внешним источником или ошибкой в вычислениях.

```
void (*signal(int sig, void (*handler)(int)))(int)
```

signal устанавливает, как будут обрабатываться последующие сигналы. Если параметр *handler* имеет значение **SIG_DFL**, то используется зависимая от реализации "обработка по умолчанию"; если значение *handler* равно **SIG_IGN**, то сигнал игнорируется; в остальных случаях будет выполнено обращение к функции, на которую указывает *handler* с типом сигнала в качестве аргумента. В число допустимых видов сигналов входят:

SIGABRT	- аварийное завершение, например от *abort*;
SIGFPE	- арифметическая ошибка: деление на 0 или переполнение;
SIGILL	- неверный код функции (недопустимая команда);
SIGINT	- запрос на взаимодействие, например прерывание;
SIGSEGV	- неверный доступ к памяти, например выход за границы;
SIGTERM	- требование завершения, посланное в программу.

signal возвращает предыдущее значение *handler* в случае специфицированного сигнала, или **SIGERR** в случае возникновения ошибки.

Когда в дальнейшем появляется сигнал *sig*, сначала восстанавливается готовность поведения "по умолчанию", после чего вызывается функция, заданная в параметре *handler*, т.е. как бы выполняется вызов (**handler*)(sig). Если функция *handler* вернет управление назад, то вычисления возобновятся с того места, где застал программу пришедший сигнал. Начальное состояние сигналов зависит от реализации.

```
int raise(int sig)
```

raise посылает в программу сигнал *sig*. В случае неудачи возвращает ненулевое значение.

B10. Функции даты и времени: <time.h>

Заголовочный файл <**time.h**> объявляет типы и функции, связанные с датой и временем. Некоторые функции имеют дело с местным временем, которое может отличаться от календарного, например в связи с зонированием времени. Типы **clock_t** и **time_t** - арифметические типы для представления времени, а *struct tm* содержит компоненты календарного времени:

```
int tm_sec;   - секунды от начала минуты (0,61); -- I.B.: все же наверно от 0 до 59
```

```
int tm_min;   - минуты от начала часа (0,59);
int tm_hour;  - часы от полуночи (0,23);
int tm_mday;  - число месяца (1,31);
int tm_mon;   - месяцы с января(0,11);
int tm_year;  - годы с 1900;
int tm_wday;  - дни с воскресенья (0,6);
int tm_yday;  - дни с 1 января (0,365);
int tm_isdst; - признак летнего времени.
```

Значение *tm_isdst* - положительное, если время приходится на сезон, когда время суток сдвинуто на 1 час вперед, нуль в противном случае и отрицательное, если информация не доступна.

```
clock_t clock(void)
```

clock возвращает время, фиксируемое процессором от начала выполнения программы, или -1, если оно не известно. Для выражения этого времени в секундах применяется формула clock()/CLOCKS_PER_SEC.

```
time_t time(time_t *tp)
```

time возвращает текущее календарное время (т. е. время, прошедшее после определенной даты, - обычно после 0 ч 00 мин 00 с GMT 1-го января 1970 г. - примеч. ред.) или -1, если время не известно. Если *tp* не равно NULL, то возвращаемое значение записывается и в **tp*.

```
double difftime(time_t time2, time_t time1)
```

difftime возвращает разность *time2 - time1*, выраженную в секундах.

```
time_t mktime(struct tm *tp)
```

mktime преобразует местное время, заданное структурой **tp*, в календарное, выдавая его в том же виде, что и функция *time*. Компоненты будут иметь значения в указанных диапазонах. Функция *mktime* возвращает календарное время или -1, если оно не представимо.

Следующие четыре функции возвращают указатели на статические объекты, каждый из которых может быть изменен другими вызовами.

```
char *asctime(const struct tm *tp)
```

asctime переводит время в структуре **tp* в строку вида

```
    Sun Jan 3 15:14:13 1988\n\0
```

```
char *ctime(const time_t *tp)
```

ctime переводит календарное время в местное, что эквивалентно выполнению asctime(localtime(tp))

```
struct tm *gmtime(const time_t *tp)
```

gmtime переводит календарное время во Всемирное координированное время (Coordinated Universal Time - UTC). Выдаст NULL, если UTC не известно. Имя этой функции, *gmtime*, происходит от Greenwich Mean Time (среднее время по Гринвичскому меридиану).

```
struct tm *localtime(const time_t *tp)
```

localtime переводит календарное время **tp* в местное.

```
size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp)
```

strftime форматирует информацию о дате и времени из **tp* в строку *s* согласно формату *fmt*, который имеет много общих черт с форматом, задаваемым в функции *printf*. Обычные символы (включая и завершающий символ '\0') копируются в *s*. Каждая пара, состоящая из **%** и буквы, заменяется, как показано ниже, с использованием значений по форме, соответствующей местным традициям. В *s* размещается не более *smax* символов; *strftime* возвращает число символов без учета '\0' или нуль, если число сгенерированных символов больше *smax*.

%a	сокращенное название дня недели
%A	полное название дня недели

%b	сокращенное название месяца
%B	полное название месяца
%c	местное представление даты и времени
%d	день месяца (01-31)
%H	час (24-часовое время) (00-23)
%I	час (12-часовое время) (01-12)
%j	день от начала года (001-366)
%m	месяц (01-12)
%M	минута (00-59)
%p	местное представление AM или PM (до или после полудня)
%S	секунда (00-61)
%U	неделя от начала года (считая, что воскресенье - 1-й день недели) (00-53)
%w	день недели (0-6, номер воскресенья - 0)
%W	неделя от начала года (считая, что понедельник - 1-й день недели) (00-53)
%x	местное представление даты
%X	местное представление времени
%y	год без указания века (00-99)
%Y	год с указанием века
%Z	название временной зоны, если она есть
%%	%

B11. Зависящие от реализации пределы: <limits.h> и <float.h>

Заголовочный файл **<limits.h>** определяет константы для размеров целочисленных типов. Ниже перечислены минимальные приемлемые величины, но в конкретных реализациях могут использоваться и большие значения.

CHAR_BIT	8	битов в значении **char**
SCHAR_MAX	**UCHAR_MAX** или **SCHAR_MAX**	максимальное значение *char*
CHAR_MIN	0 или **CHAR_MIN**	минимальное значение *char*
INT_MAX	+32767	максимальное значение *int*
INT_MIN	-32767 (I.B.:*обычно это значение -32768*)	минимальное значение *int*
LONG_MAX	+2147463647	максимальное значение *long*
LONG_MIN	-2147483647 (I.B.:*обычно это значение -2147483648*)	минимальное значение *long*
SCHAR_MAX	+127	максимальное значение *signed char*
SCHAR_MIN	-127 (I.B.:*обычно это значение -128*)	минимальное значение *signed char*
SHRT_MAX	+32767	максимальное значение *short*

SHRT_MIN	-32767 (I.B.:*обычно это значение -32768*)	минимальное значение *short*
UCHAR_MAX	255	максимальное значение *unsigned char*
UINT_MAX	65535	максимальное значение *unsigned int*
ULONG_MAX	4294967295	максимальное значение *unsigned long*
USHRT_MAX	65535	максимальное значение *unsigned short*

Имена, приведенные в следующей таблице, взяты из <**float.h**> и являются константами, имеющими отношение к арифметике с плавающей точкой. Значения (если они есть) представляют собой минимальные значения для соответствующих величин. В каждой реализации устанавливаются свои значения.

FLT_RADIX	2	основание для представления порядка, например: 2, 16
FLT_ROUNDS		способ округления при сложении чисел с плавающей точкой
FLT_DIG	6	количество верных десятичных цифр
FLT_EPSILON	1E-5	минимальное x, такое, что 1.0 + x != 1.0
FLT_MANT_DIG		количество цифр по основанию **FLT_RADIX** в мантиссе
FLT_MAX	1E+37	максимальное число с плавающей точкой
FLT_MAX_EXP		максимальное n, такое, что **FLT_RADIX**^{n}-1 представимо
FLT_MIN	1E-37	минимальное нормализованное число с плавающей точкой
FLT_MIN_EXP		минимальное n, такое, что 10^{n} представимо в виде нормализованного числа
DBL_DIG	10	количество верных десятичных цифр для типа *double*
DBL_EPSILON	1E-9	минимальное x, такое, что 1.0 + x != 1.0, где x принадлежит типу *double*
DBL_MANT_DIG		количество цифр по основанию **FLT_RADIX** в мантиссе для чисел типа *double*
DBL_MAX	1E+37	максимальное число с плавающей точкой типа *double*
DBL_MAX_EXP		максимальное n, такое, что **FLT_RADIX**^{n}-1 представимо в виде числа типа *double*
DBL_MIN	1E-37	минимальное нормализованное число с плавающей точкой типа *double*
DBL_MIN_EXP		минимальное n, такое, что 10^{n} представимо в виде нормализованного числа типа *double*

[Назад] [Содержание] [Вперед]

[Главная] [Гостевая]