КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАМИРОВАНИЯ

Лекция № 16 — Библиотека языка С

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2021

Оглавление

Библиотека

Строка — это непрерывная последовательность символов, оканчивающаяся первым нулевым символом и включающая его.

Вместо этого термина иногда используется термин *многобайтовая строка* , чтобы подчеркнуть специальную обработку многобайтовых символов, содержащихся в строке, или чтобы избежать путаницы с широкой строкой (wide string).

Указатель на строку — это указатель на ее начальный (с наименьшим адресом) символ.

Длина строки — это число байтов, предшествующих нулевому символу.

Значение строки — это последовательность значений содержащихся в строке символов в естественном порядке.

Символ десятичной запятой — это символ, используемый функциями, которые преобразуют числа с плавающей запятой в или из последовательностей символов, чтобы обозначить начало дробной части таких последовательностей символов. Может быть изменен функцией **`setlocale`**.

Нулевой широкий символ — это широкий символ со значением кода ноль.

Широкая строка — это непрерывная последовательность широких символов, оканчивающаяся и включающая первый нулевой широкий символ.

Указатель на широкую строку — это указатель на ее начальный (с наименьшим адресом) широкий символ.

Длина широкой строки — это число широких символов, предшествующих нулевому широкому символу.

Значение широкой строки — это последовательность значений кодов широких символов, содержащихся по порядку в широкой строке.

Последовательность сдвига — это непрерывная последовательность байтов в многобайтовой строке, которая (потенциально) вызывает изменение сдвигового состояния анализатора (shift state).

Многобайтовый набор символов может иметь кодирование, специфичное для конкретной локали.

Стандартные заголовки

Каждая библиотечная функция объявляется с типом, который включает в себя прототип.

Библиотечная функция объявляется в заголовке, содержимое которого доступно с помощью директивы препроцессора **`#include`**.

Заголовок объявляет набор связанных друг с другом функций, а также любые необходимые типы и дополнительные макросы, необходимые для облегчения их использования.

Объявления типов обычно не включают квалификаторы типов, хотя есть исключения.

Стандартные заголовки[1]:

<assert.h>	<inttypes.h>	<signal.h>	<stdint.h>	<threads.h>
<complex.h>	<iso646.h>	<stdalign.h>	<stdio.h>	<time.h>
<ctype.h>	<limits.h>	<stdarg.h>	<stdlib.h>	<uchar.h>
<errno.h>	<locale.h>	<stdatomic.h>	<stdnoreturn.h>	<wchar.h>
<fenv.h>	<math.h>	<stdbool.h>	<string.h>	<wctype.h>
<float.h>	<setjmp.h>	<stddef.h>	<tgmath.h>	

Стандартные заголовки могут включаться в любом порядке — каждый из них может быть включен более одного раза в данную область, при этом эффект будет аналогичен тому, как если бы заголовок был включен только один раз.

---

1 Заголовки <complex.h>, <stdatomic.h> и <threads.h> представляют собой условные возможности и некоторые реализации их не поддерживают.

Защита (guard) во включаемых файлах

```
/* stdint.h */
#ifndef _STDINT_H
#define _STDINT_H 1

#define __GLIBC_INTERNAL_STARTING_HEADER_IMPLEMENTATION
#include <bits/libc-header-start.h>
#include <bits/types.h>
#include <bits/wchar.h>
#include <bits/wordsize.h>
...
#endif /* _STDINT_H */
```

Есть исключение — эффект от включения **<assert.h>** зависит от определения макроса **NDEBUG**.

Заголовок, если он используется, должен быть включен перед первой ссылкой на любую из функций или объектов, которые он объявляет, или на любой из типов или макросов, которые он определяет.

Однако, если идентификатор объявлен или определен в более чем одном заголовке, второй и последующие ассоциированные с этим идентификатором заголовки могут быть включены после самой первой ссылки на этот идентификатор.

Любое определение объектоподобного макроса расширяется до кода, который при необходимости полностью защищается скобками таким образом, чтобы он в произвольном выражении образовывал группу, как если бы это был один идентификатор.

Все объявления библиотечных функций имеют внешний тип связывания.

Зарезервированные идентификаторы

Некоторые заголовки объявляют или определяют ряд идентификаторов, которые всегда зарезервированы либо для любого использования, либо в области видимости файла.

1) Все идентификаторы, которые начинаются со знака подчеркивания и следующей за ним заглавной буквы или другого подчеркивания, всегда зарезервированы для любого использования.

2) Все идентификаторы, которые начинаются с подчеркивания, всегда зарезервированы для использования в качестве идентификаторов с областью действия файла как в обычном пространстве, так и в пространстве имен тегов.

3) Каждое имя макроса зарезервировано для использования, если включен какой-либо из связанных с ним заголовков.

4) Все идентификаторы с внешней связью и **`errno`** всегда зарезервированы для использования в качестве идентификаторов с внешней связью.

5) Каждый идентификатор с областью действия файла зарезервирован для использования в качестве имени макроса и в качестве идентификатора с областью действия файла в том же пространстве имен, если включен какой-либо из связанных с ним заголовков.

Другие идентификаторы не зарезервированы.

Если программа объявляет или определяет идентификатор в контексте, в котором он зарезервирован, или определяет зарезервированный идентификатор как имя макроса, поведение не определено.

Если программа удаляет (с помощью **`#undef`**) любое макроопределение зарезервированного идентификатора, поведение не определено.

Использование библиотечных функций

Стандарт говорит, что в целом для библиотечных функций справедливо, если явно не указано иное в подробных описаниях, следующее:

- если аргумент функции имеет недопустимое значение (например, указатель вне адресного пространства программы, или нулевой указатель, или указатель на немодифицируемую память в том случае, когда соответствующий параметр не квалифицирован как **`const`**), или тип, не ожидаемые функцией с переменным числом аргументов, поведение не определено.

- если аргумент функции описывается как массив, указатель, фактически передаваемый функции, должен указывать на первый элемент такого массива.

- функции в стандартной библиотеке не являются гарантированно *реентерабельными* и могут изменять объекты со статическим или потоковым временем существования.

- если в подробных описаниях явно не указано иное, библиотечные функции *не подвержены гонкам* данных.

- в тех случаях, когда объекты не являются видимыми для пользователей (opaque) и защищены от гонок данных, реализации обычно используют в потоках свои собственные внутренние объекты.

Состояние гонки (race condition)

Состояние гонки — возможность асинхронных изменений состояния системы/программы в процессе параллельного выпонения нескольких ветвей.

Например, есть два параллельно выполняющихся процесса P и Q, имеющие доступ к переменным x и y. Процессы выполняют следующие операции:

```
P: x = 2        Q: x = 3
   y = x – 1       y = x + 1
```

В данном случае имеет место состязание процессов (race condition) за вычисление значений переменных x и y.

Реентерабельность

Компьютерная программа в целом или её отдельная процедура называется реентерабельной (от англ. *reenter* — повторный вход), если она разработана таким образом, что одна и та же копия инструкций программы в памяти может быть совместно использована несколькими потоками управления (нитями или процессами). При этом второй поток может вызвать реентерабельный код до того, как с ним завершит работу первый и это, как минимум, не должно приводить к ошибкам, а при корректной реализации не должно вызвать потери вычислений (то есть не должно появиться необходимости выполнять уже выполненные фрагменты кода).

Для обеспечения реентерабельности необходимо выполнение нескольких условий:

- никакая часть вызываемого кода не должна модифицироваться;
- вызываемая процедура не должна сохранять информацию между вызовами;
- если процедура изменяет какие-либо данные, то они должны быть уникальными для каждого потока управления;
- процедура не должна возвращать указатели на объекты, общие для разных потоков.

В общем случае, для обеспечения реентерабельности необходимо, чтобы вызывающий процесс или функция каждый раз передавал вызываемому процессу все необходимые данные. Таким образом, функция, которая зависит только от своих параметров, не использует глобальные и статические переменные и вызывает только реентерабельные функции, будет реентерабельной.

Если функция использует глобальные или статические переменные, необходимо обеспечить, чтобы каждый пользователь хранил свою локальную копию этих переменных.

Пример использования функции

Функция **`atoi`** может быть использована любым из нескольких способов:

1) используя связанный с ней заголовок (возможно, генерируя при этом макрорасширение)

```
#include <stdlib.h>
const char *str;
/* ... */
i = atoi(str);
```

2) путем использования связанного с ней заголовка (гарантированно генерирующего истинную ссылку на функцию)

```
#include <stdlib.h>
const char *str;
/* ... */
i = (atoi)(str);
```

3) с помощью явного объявления

```
extern int atoi(const char *);
const char *str;
/* ... */
i = atoi(str);
```

Диагностика <assert.h>

assert — прекращает работу программы при ложном утверждении.

```
#include <assert.h>
void assert(scalar expression);
```

Данный макрос предназначен для написания *формальных утверждений корректности* (assertions)[2], что помогает программистам находить ошибки в своих программах или обрабатывать исключительные случаи посредством завершения программы, при котором выводится немного отладочной информации.

Если **expression** ложно (т. е., при сравнении равно нулю), то **assert()** печатает сообщение об ошибке в стандартный поток ошибок и завершает программу вызовом **abort(3)**.

Сообщение об ошибке содержит имя файла и функцию, содержащую вызов **assert()**, номер строки исходного кода вызова и текст аргумента; пример:

```
prog: some_file.c:16: some_func: Assertion `val == 0' failed.
```

Ничего не возвращается.

Если определён макрос **NDEBUG** на момент включения последнего **<assert.h>**, то макрос **assert()** не генерирует код, и, следовательно ничего вызывает. Не рекомендуется определять **NDEBUG**, если **assert()** используется для обнаружения ошибок условий, так как ПО может повести себя непредсказуемо.

2 Один из аспектов парадигмы контрактного программирования

Общие определения <stddef.h>

Заголовок **<stddef.h>** определяет несколько макросов и объявляет несколько типов. Некоторые из них также определяются в других заголовках.

Типы

ptrdiff_t

является целочисленным типом со знаком, в котором представляется результат вычитания двух указателей;

size_t

целый тип без знака в котором представляется результат оператора **sizeof**.

max_align_t

это объектный тип, выравнивание которого настолько велико, насколько это поддерживается реализацией во всех контекстах;

wchar_t

является целочисленным типом, диапазон значений которого может представлять различные коды для всех элементов самого большого из расширенных наборов символов всех поддерживаемых локалей.

Нулевой символ имеет кодовое значение, равное нулю.

Если реализация не определяет макрос **__STDC_MB_MIGHT_NEQ_WC__**, каждый элемент базового набора символов имеет кодовое значение, равное его значению, когда он используется как одиночный символ в целочисленной символьной константе.

Макросы

`NULL`

расширяется до целочисленного константного выражения, обозначающего значение нулевого указателя.

`offsetof(type, имя-члена)`

расширяется до целочисленного константного выражения, имеющего тип **`size_t`**, значением которого является смещение в байтах, до элемента структуры с именем **`имя-члена`** от начала его структуры (обозначенного как **`type`**).

Тип и обозначение элемента должны быть такими, чтобы для

```
static type t;
```

выражение **`&(t.member)`** вычислялось как адресная константа.

Если указанный член является битовым полем, поведение не определено (для битового поля нельзя получить адрес).

Реализация *может не поддерживать* объекты достаточной величины, чтобы типы **`size_t`** и **`ptrdiff_t`** имели целочисленный ранг преобразования выше, чем для типа **`signed long int`**.

Общие утилиты <stdlib.h>

Заголовок **<stdlib.h>** объявляет пять типов и несколько функций общего назначения и определяет несколько макросов.

Объявляются следующие типы: **size_t** и **wchar_t**,

div_t

структурный тип, который является типом значения, возвращаемого функцией **div**,

ldiv_t

структурный тип, который является типом значения, возвращаемого функцией **ldiv**,

lldiv_t

структурный тип, который является типом значения, возвращаемого функцией **lldiv**,.

Макросы:

NULL
EXIT_FAILURE
EXIT_SUCCESS

расширяются до целочисленных константных выражений, которые могут быть использованы в качестве аргумента функции выхода с целью возврата статуса неудачного или успешного завершения, в хост-среду.

RAND_MAX

который расширяется до целочисленного константного выражения, являющегося максимальным значением, возвращаемым функцией **rand( )**; а также

MB_CUR_MAX

который расширяется до положительного целочисленного выражения с типом **size_t**, являющимся максимальным числом байтов в многобайтовом символе в случае расширенного набора символов, заданного текущим языковым стандартом (категория **LC_CTYPE**), который никогда не превышает **MB_LEN_MAX**.

Функции преобразования целых чисел

Функции **atof**, **atoi**, **atol** и **atoll** не влияют на значение целочисленного выражения **errno** в случае ошибки.

Если значение результата не может быть представлено, поведение не определено.

```
#include <stdlib.h>

double atof(const char *nptr);
int    atoi(const char *nptr);
long   atol(const char *nptr);
long   long atoll(const char *nptr);
```

atof – преобразует строку в значение типа **double**

atoi – преобразует строку в значение типа **int**

atol – преобразует строку в значение типа **long**

atoll – преобразует строку в значение типа **long long**

```
strtod(nptr, (char **)NULL) // atof
```

strtod, **strtof**, **strtold** — преобразует строку ASCII в число с плавающей точкой

```
#include <stdlib.h>

double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

Функции **strtod()**, **strtof()** и **strtold()** преобразуют начальную часть строки, на которую указывает **nptr**, в числа типа **double**, **float**, и **long double**, соответственно.

Ожидаемый вид строки (её начальная часть) — это начальные пробельные символы (необязательно), распознаваемые функцией **isspace(3)**, возможно знаки плюс ('**+**') или минус ('**-**'), а затем либо (а) десятичное число, либо (б) шестнадцатеричное число, либо (в) бесконечность, либо (г) **NAN** (not-a-number, нечисловое значение).

Десятичное число состоит из непустой последовательности десятичных цифр от 0 до 9, возможно содержащей символ дробного разделителя (десятичная точка, зависит от настройки локали; обычно это символ точки '**.**'), возможно с последующей десятичной экспонентой.

Десятичная экспонента состоит из символа '**E**' или '**e**', далее возможен знак плюс или минус, а затем непустая последовательность десятичных цифр, означающая умножение всей начальной части на 10 в указанной степени.

Шестнадцатеричное число состоит из символов «**0x**» или «**0X**» с последующей непустой последовательностью из шестнадцатеричных цифр, возможно содержащей символ дробного разделителя, далее может следовать двоичная экспонента.

Двоичная экспонента состоит из символа '**P**' или '**p**', далее, возможно, знака плюс или минус, и непустой последовательности десятичных цифр, обозначающих умножение всего начального числа на 2 в указанной степени. По меньшей мере должен быть указан либо символ дробного разделителя либо двоичная экспонента.

Бесконечность — это либо значение «**INF**» либо «**INFINITY**», регистр символов не учитывается.

Нечисловое значение (**NAN**) — это значение «**NAN**» (регистр символов не учитывается) возможно сопровождаемое (последовательность **n** символов), где последовательность **n** символов, определяет тип нечислового значения в зависимости от текущего представления **NAN** в системе.

Возвращаемое значение

Функции возвращают преобразованное значение, если таковое существует.

Если **endptr** не равно **NULL**, то указатель на символ, следующий за последним обработанным символом, сохраняется в место, указываемое **endptr**.

Если никаких преобразований не производилось, то возвращается ноль, а значение **nptr** сохраняется в той позиции, на которую ссылается **endptr** (если **endptr** не равно **NULL**).

Если правильное значение вызвало бы переполнение, то возвращается **HUGE_VAL** (**HUGE_VALF**, **HUGE_VALL**) (в зависимости от знака величины), а в переменную **errno** записывается **ERANGE**.

Так как возвращаемым значением может быть 0 как при успешном выполнении, так и в случае ошибки, вызывающая программа должна присвоить **errno** значение 0 до вызова, а после вызова определить возникновение ошибки по ненулевому значению **errno**.

strtol, **strtoll**, **strtoq** – преобразует строку в длинное целое число

```
#include <stdlib.h>

long int strtol(const char *nptr,
                char **endptr,
                int base);

long long int strtoll(const char *nptr,
                      char **endptr,
                      int base);
```

Функция **strtol()** преобразует начальную часть строки **nptr** в длинное целое число согласно системе счисления **base**, значение которой может быть от 2 до 36 включительно или равно специальному значению 0.

Строка может начинаться с произвольного количества пробельных символов (определяемых при помощи **isspace(3)**), затем может быть указан знак «**+**» или «**-**».

Если **base** равно 0 или 16 и строка начинается с префикса «**0x**» или «**0X**», это означает использование шестнадцатеричной системы исчисления.

Если **base** равно нулю, то используется десятичная система счисления.

Если последующий символ также равен «**0**», используется восьмеричная система исчисления).

Остаток строки преобразуется в число с типом **long int**. Этот процесс останавливается, если в строке встречается некорректный символ для указанной системе счисления.

В системах счисления больших 10, символ «A» в верхнем или нижнем регистре означает 10, «B» означает 11 и так далее до «Z», означающего 35.

Если значение **endptr** не NULL, то **strtol()** записывает адрес первого некорректного символа в ***endptr**.

Если в строке вообще нет цифр, то **strtoul()** сохраняет изначальное значение **nptr** в ***endptr** и возвращает 0. В частности, если ***nptr** не равно '**\0**', но ****endptr** равно '**\0**' при возврате, то вся строка состоит из корректных символов.

Функция **strtoll()** работает так же, как и **strtol()**, но возвращает число с типом **long long int**.

Возвращаемое значение

Функция **strtol()** возвращает результат преобразования, если не возникают переполнения или исчерпания. Если возникает исчерпание, то **strtol()** возвращает **LONG_MIN**. Если возникает переполнение, то **strtoul()** возвращает **LONG_MAX**. В обоих случаях переменной **errno** присваивается значение **ERANGE**.

То же самое относится к **strtoll()**, только вместо **LONG_MIN** и **LONG_MAX** возвращается **LLONG_MIN** и **LLONG_MAX**.

Ошибки

EINVAL (нет в C99) — аргумент **base** содержит неподдерживаемое значение.

ERANGE — результат вне диапазона.

Реализация может также устанавливать **errno** в **EINVAL** в случае, когда преобразование не было выполнено (не было встречено цифр и возвращён 0).

Так как **strtol()** может обоснованно вернуть 0, **LONG_MAX** или **LONG_MIN** (а **strtoll()** — **LLONG_MAX** или **LLONG_MIN**) как при успешном выполнении, так и в случае ошибки, вызывающая программа до вызова должна присвоить **errno** значение 0, а после вызова определить возникновение ошибки по ненулевому значению **errno**.

Согласно POSIX.1, в локалях отличных от «C» и «POSIX», эти функции могут преобразовывать другие, определяемые реализацией, строки с числами.

Пример

Программа, представленная далее, показывает использование **strtol()**. В первом аргументе командной строки указывается срока, из которой **strtol()** должна извлечь число. Во втором (необязательном) аргументе указывается система счисления, используемая для преобразования (этот аргумент преобразуется в число с помощью **atoi(3)**, функции, которая не учитывает ошибки и имеет более простой интерфейс по сравнению с **strtol()**).

Вот несколько результатов работы этой программы:

Исходный код программы

```
#include <stdlib.h>
#include <limits.h>
#include <stdio.h>
#include <errno.h>

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

    int base;
    char *endptr, *str;
    long val;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s строка [система_счисления]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    str = argv[1];
    base = (argc > 2) ? atoi(argv[2]) : 10;

    errno = 0;    // чтобы выявить ошибку после вызова
    val = strtol(str, &endptr, base);
```

```
    // проверка возможных ошибок
    if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN)) ||
        (errno != 0 && val == 0)) {
        perror("strtol");
        exit(EXIT_FAILURE);
    }

    if (endptr == str) {
        fprintf(stderr, "Цифры отсутствуют\n");
        exit(EXIT_FAILURE);
    }

    // если мы дошли сюда, то strtol() успешно преобразовала число
    printf("strtol() вернула %ld\n", val);

    if (*endptr != '\0')        // необязательно ошибка …
        printf("Остальные символы после числа: %s\n", endptr);

    exit(EXIT_SUCCESS);
}
```

```
$ gcc prog.c
$ ./a.out 123
strtol() вернула 123
$ ./a.out '    123'
strtol() вернула 123
$ ./a.out 123abc
strtol() вернула 123
Остальные символы после числа: abc
$ ./a.out 123abc 55
strtol: некорректный аргумент
$ ./a.out ''
Цифры отсутствуют
$ ./a.out 4000000000
strtol: Числовой результат выходит за диапазон
```

Функции управления памятью

`malloc`, **`free`**, **`calloc`**, **`realloc`** — выделяют и освобождают динамическую память;
`aligned_alloc` — выделяет выровненную память.

Порядок и непрерывность памяти, выделенной последовательными вызовами функций **`aligned_alloc`**, **`calloc`**, **`malloc`** и **`realloc`**, не определены.

Указатель, возвращаемый в случае успешного выделения, выравнивается соответствующим образом, чтобы его можно было назначить указателю на любой тип объекта с фундаментальным требованием выравнивания, а затем использовать для доступа к такому объекту или массиву таких объектов в выделенном пространстве (до тех пор, пока пространство не будет явно освобождено).

Время жизни выделенного объекта простирается от выделения до освобождения.

Каждое такое распределение дает указатель на объект, не пересекающийся ни с каким другим объектом.

Возвращенный указатель указывает на начало (младший байтовый адрес) выделенного пространства.

Если пространство не может быть выделено, возвращается нулевой указатель.

Если размер запрошенного пространства равен нулю, поведение определяется реализацией, например возвращается нулевой указатель.

Выделение области памяти

malloc, **free**, **calloc**, **realloc** – распределяют и освобождают динамическую память

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

Функция **malloc()** распределяет **size** байтов и возвращает указатель на распределённую память. Память при этом не инициализируется.

Если значение **size** равно 0, то **malloc()** возвращает или **NULL**, или уникальный указатель, который можно без опасений передавать **free()**.

Функция **free()** освобождает место в памяти, указанное в **ptr**, которое должно быть получено ранее вызовом функции **malloc()**, **calloc()** или **realloc()**.

В противном случае (или если вызов **free(ptr)** уже выполнялся) дальнейшее поведение не определено. Если значение **ptr** равно **NULL**, то не выполняется никаких действий.

Функция **calloc()** распределяет память для массива размером **nmemb** элементов по **size** байтов каждый и возвращает указатель на распределённую память. Данные в выделенной памяти при этом обнуляются. Если значение **nmemb** или **size** равно **0**, то **calloc()** возвращает или **NULL**, или уникальный указатель, который можно без опасений передавать **free()**.

Функция **realloc()** меняет размер блока памяти, на который указывает **ptr**, на размер, равный **size** байт. Содержимое памяти не будет изменено от начала области в пределах наименьшего из старого и нового размеров.

Если новый размер больше старого, то добавленная память не будет инициализирована.

Если значение **ptr** равно **NULL**, то вызов эквивалентен **malloc(size)** для всех значений **size**.

Если значение **size** равно нулю и **ptr** не равно **NULL**, то вызов эквивалентен **free(ptr)**.

Функции **malloc()** и **calloc()** возвращают указатель на распределённую память, выровненную должным образом для любого **встроенного типа**.

При ошибке возвращается **NULL**. Значение **NULL** также может быть получено при успешной работе вызова **malloc()**, если значение **size** равно нулю, или **calloc()** — если значение **nmemb** или **size** равно нулю.

Функция **free()** ничего не возвращает.

Функция **realloc()** возвращает указатель на новую распределённую память, выровненную должным образом для любого встроенного типа. Возвращаемый указатель может отличаться от **ptr**, или равняться **NULL**, если запрос завершился с ошибкой.

Если значение **size** было равно нулю, то возвращается либо **NULL**, либо указатель, который может быть передан **free()**.

Если **realloc()** завершилась с ошибкой, то начальный блок памяти остаётся нетронутым — он ни освобождается, ни или

Ошибки

Функции **calloc()**, **malloc()**, **realloc()** могут завершаться со следующей ошибкой:

ENOMEM — не хватает памяти. В этом случае скорее всего приложением достигнут лимит **RLIMIT_AS** или **RLIMIT_DATA**, описанный в **getrlimit(2)**.

Выделение выровненной области памяти

```
#include <stdlib.h>

void *aligned_alloc(size_t alignment, size_t size);
```

Функция **aligned_alloc()** выделяет **size** байт и возвращает указатель на выделенную память.

Адрес, соответствующий указателю, будет кратен значению **alignment**, которое должно быть степенью двойки и кратно **sizeof(void *)**.

Если **size** равно 0, то значение указателя равно **NULL**, или является уникальным значением, который позднее можно передать в **free(3)**.

Эта функция не обнуляет выделяемую память.

Функция **aligned_alloc()**, возвращает **NULL** при ошибках.

Ошибки

EINVAL — аргумент **alignment** не является степенью двойки или не кратен **sizeof(void *)**.

ENOMEM — недостаточно памяти для выполнения запроса о выделении.

Целочисленные типы <cstdint> (stdint.h)

Заголовок **<stdint.h>** объявляет наборы целочисленных типов, имеющих указанную ширину, и определяет соответствующие наборы макросов.

Он также определяет макросы, которые определяют ограничения целочисленных типов, соответствующие типам, определенным в других стандартных заголовках.

Типы определены в следующих категориях:

- целочисленные типы, имеющие определенную точную ширину;
- целочисленные типы, имеющие по крайней мере определенную указанную ширину;
- самые быстрые целочисленные типы, имеющие по крайней мере определенную указанную ширину;
- целочисленные типы, достаточно широкие, чтобы содержать указатели на объекты;
- целочисленные типы, имеющие наибольшую ширину.

(Некоторые из этих типов могут обозначать один и тот же тип.)

Соответствующие макросы определяют предельные значения объявленных типов и создают подходящие константы.

Для каждого типа из описанных в **<stdint.h>** стандарта, конкретная реализация объявляет это имя как **typedef** и определяет связанные с ним макросы.

Целочисленные типы

Определения typedef, отличающиеся только отсутствием или наличием начального **u**, обозначают соответствующие типы со знаком и без знака. Если реализация обеспечивает один из этих соответствующих типов, она также предоставляет и другой.

Целочисленные типы точной ширины

Имя typedef **intN_t** обозначает целочисленный тип со знаком с шириной **N**, без дополнительных битов и представлением в виде дополнения до двух. Таким образом, **int8_t** обозначает такой целочисленный тип со знаком шириной ровно 8 бит.

Определяются следующие обязательные типы:

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

typedef -имя **uintN_t** обозначает целочисленный тип без знака с шириной **N** и без битов заполнения. Таким образом, **uint24_t** обозначает такой целочисленный тип без знака с шириной ровно 24 бита.

Эти типы не являются обязательными. Однако, если реализация предоставляет целочисленные типы с шириной 8, 16, 32 или 64 бита, без битов заполнения и (для типов со знаком), которые имеют представление дополнения до двух, она должна определять соответствующие имена **typedef**.

Целочисленные типы минимальной ширины

Typedef-имя **`int_leastN_t`** обозначает целочисленный тип со знаком шириной не менее **`N`**, так что ни один целочисленный тип со знаком меньшего размера не имеет указанной ширины. Таким образом, **`int_least32_t`** обозначает целочисленный тип со знаком шириной не менее 32 бит.

Typedef-имя **`uint_leastN_t`** обозначает целочисленный тип без знака с шириной не менее **`N`**, так что целочисленный тип без знака с меньшим размером не имеет по крайней мере указанной ширины. Таким образом, **`uint_least16_t`** обозначает целочисленный тип без знака с шириной не менее 16 бит.

Определяются следующие обязательные типы:

```
int_least8_t      uint_least8_t
int_least16_t     uint_least16_t
int_least32_t     uint_least32_t
int_least64_t     uint_least64_t
```

Все остальные типы данной формы являются необязательными.

Самые быстрые целочисленные типы минимальной ширины

Каждый из следующих типов обозначает целочисленный тип, который обычно является самым быстрым[3] в выполнении операций среди всех целочисленных типов, которые имеют по меньшей мере указанную ширину.

Typedef-имя **int_fastN_t** обозначает самый быстрый целочисленный тип со знаком и шириной не менее **N**. Typedef-имя **uint_fastN_t** обозначает самый быстрый целочисленный тип без знака с шириной не менее **N**.

Определяются следующие обязательные типы:

```
int_fast8_t      uint_fast8_t
int_fast16_t     uint_fast16_t
int_fast32_t     uint_fast32_t
int_fast64_t     uint_fast64_t
```

Все остальные типы данной формы являются необязательными.

3 262) не гарантируется, что обозначенный тип является самым быстрым во всех случаях; если реализация не имеет четких оснований для выбора одного типа из нескольких, она просто выберет некоторый целочисленный тип, удовлетворяющий требованиям знаковости и ширины.

Целочисленные типы, способные содержать указатели объектов

intptr_t – тип обозначает целочисленный тип со знаком и со свойством, что любой действительный указатель на **void** может быть преобразован в этот тип, а затем преобразован обратно в указатель на **void**, и результат будет совпадать с исходным указателем:

uintptr_t – тип обозначает целочисленный тип без знака со свойством, что любой действительный указатель на **void** может быть преобразован в этот тип, затем преобразован обратно в указатель на **void**, и результат будет совпадать с исходным указателем:

Эти типы не являются обязательными.

Целочисленные типы наибольшей ширины

intmax_t – тип обозначает целочисленный тип со знаком, способный представлять любое значение любого целочисленного типа со знаком.

uintmax_t – тип обозначает целочисленный тип без знака, способный представлять любое значение любого целочисленного типа без знака:

Эти типы являются обязательными.

Предельные значения целочисленных типов указанной ширины

Следующие объектоподобные макросы определяют минимальное и максимальное ограничения типов, объявленных в **<stdint.h>**.

Каждый макрос представляет константное выражение, подходящее для использования в директивах препроцессора **#if**, и это выражение имеет тот же тип, что и выражение, являющееся объектом соответствующего типа, преобразованным в соответствии с целочисленными преобразованиями. Его значение, определяемое реализацией, должно быть равно или больше по величине (абсолютное значение), чем соответствующее значение, приведенное ниже, с тем же знаком, за исключением случаев, когда указано, что оно точно соответствует данному значению.

Определения основных или расширенных целочисленных типов.

signed type	unsigned type	description
intmax_t	**uintmax_t**	Целочисленный тип максимально поддерживаемой ширины.
int8_t **int16_t** **int32_t** **int64_t**	**uint8_t** **uint16_t** **uint32_t** **uint64_t**	Целочисленный тип ширины ровно 8, 16, 32 или 64 бита. Для типов со знаком отрицательные значения представлены с использованием дополнения до 2. Нет битов заполнения. Необязательно: Эти определения типов не определяются, если не существует типов с такими характеристиками.
int_least8_t **int_least16_t** **int_least32_t** **int_least64_t**	**uint_least8_t** **uint_least16_t** **uint_least32_t** **uint_least64_t**	Целочисленный тип минимум 8, 16, 32 или 64 бит. Никаких других целочисленных типовс меньшим размером и, по крайней мере, указанной шириной не существует.
int_fast8_t **int_fast16_t** **int_fast32_t** **int_fast64_t**	**uint_fast8_t** **uint_fast16_t** **uint_fast32_t** **uint_fast64_t**	Целочисленный тип минимум 8, 16, 32 или 64 бит. По крайней мере, настолько быстрые, как любые другие целочисленные типы с, по крайней мере указанной, шириной.
intptr_t	**uintptr_t**	Целочисленный тип, способный содержать значение, преобразованное из **void** указателя, а затем преобразованное обратно в void со значением, которое совпадает с исходным. Эти определения типов на являются обязательными.

Конкретная реализация библиотеки может также определять дополнительные типы с другими значениями ширины, поддерживаемыми ее архитектурой/системой.

Таблица ограничений на значения для типов **<cstdint> / <stdint.h>**

Macro	description	defined as
INTMAX_MIN	Минимальное значение **intmax_t**	$-(2^{63}-1)$, или ниже
INTMAX_MAX	Максимальное значение **intmax_t**	$2^{63}-1$, или выше
UINTMAX_MAX	Максимальное значение **uintmax_t**	$2^{64}-1$, или выше
INTN_MIN	Мин. значение типа со знаком точной ширины	Точно $-2^{(N-1)}$
INTN_MAX	Максимальное значение типа со знаком точной ширины	Точно $2^{(N-1)}-1$
UINTN_MAX	Максимальное значение типа без знака точной ширины	Точно $2^{N}-1$
INT_LEASTN_MIN	Мин. значение типа со знаком минимальной ширины	$-(2^{(N-1)}-1)$, или ниже
INT_LEASTN_MAX	Макс. значение типа со знаком минимальной ширины	$2^{(N-1)}-1$, или выше
UINT_LEASTN_MAX	Макс. значение типа без знака минимальной ширины	$2^{N}-1$, или выше
INT_FASTN_MIN	Макс. значение быстрого типа со знаком мин. ширины	$-(2^{(N-1)}-1)$, или ниже
INT_FASTN_MAX	Макс. значение быстрого типа со знаком мин. ширины	$2^{(N-1)}-1$, или выше
UINT_FASTN_MAX	Макс.значение быстрого типа без знака мин. ширины	$2^{N}-1$, или выше
INTPTR_MIN	Минимальное значение **intptr_t**	$-(2^{15}-1)$, или ниже
INTPTR_MAX	Максимальное значение **intptr_t**	$2^{15}-1$, или выше
UINTPTR_MAX	Максимальное значение **uintptr_t**	$2^{16}-1$, или выше

Где **N** — одно из 8, 16, 32, 64 или любой другой ширины типа, поддерживаемой библиотекой.

Функциональные макросы

Данные функциональные макросы расширяются до целочисленных констант, подходящих для инициализации объектов указанных выше типов:

INTMAX_C расширяется до значения типа **intmax_t**
UINTMAX_C расширяется до значения типа **uintmax_t**
INTN_C расширяется до значения типа **int_leastN_t**
UINTN_C расширяется до значения типа **uint_leastN_t**

Например:

```
INTMAX_C(12345)  // расширяется в 12345LL или что-то подобное
```

Поддержка нескольких языков — <locale.h>

```
#include <locale.h>
```

Локаль — это сочетание языковых и культурных аспектов. Они включают в себя:

- язык сообщений;
- различные наборы символов;
- лексикографические соглашения;
- форматирование чисел;
- и прочее.

Программа должна определять локализацию и действовать согласно её установкам в целях достижения взаимосвязи различных культур.

Файл **<locale.h>** описывает типы данных, функции и макросы, необходимые для выполнения этой задачи.

В нём описаны функции: **setlocale(3)**, устанавливающая текущую локаль и **localeconv(3)**, которая возвращает информацию о форматировании чисел.

В локали существуют различные категории информации, которые программа может использовать — они описаны как макросы.

Используя их в качестве первого аргумента функции **setlocale(3)**, можно установить значение категории в желаемой локали.

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME

LC_COLLATE — эта категория определяет правила сравнения, используемые при сортировке и регулярных выражениях, включая равенство классов символов и сравнение многосимвольных элементов.

Эта категория локали изменяет поведение функций **strcoll**[4]**(3)** и **strxfrm**[5]**(3)**, которые используются для сравнения строк с учётом местного алфавита.

Например, немецкая **ßß** эсцет (sharp s) рассматривается как «ss».

4 strcoll() – сравнивает две строки с учетом настроек текущей локали

5 strxfrm() – преобразование строки с учетом локали (strcmp() \equiv strcoll())

LC_CTYPE — эта категория определяет интерпретацию последовательности байт в символы (например, одиночный или многобайтовый символ), классификацию символов (например, буквенный или цифровой) и поведение классов символов.

В системах с **glibc** эта категория также определяет правила транслитерации символов для **iconv(1)**[6] и **iconv(3)**. Она изменяет поведение функций обработки и классификации символов, таких как **isupper(3)** и **toupper(3)**, а также многобайтных символьных функций, таких как **mblen**[7]**(3)** или **wctomb**[8]**(3)**.

LC_MONETARY — эта категория определяет форматирования, используемое для денежных значений. Она изменяет информацию, возвращаемую функцией **localeconv(3)**, которая описывает способ отображения числа, например, необходимо ли использовать в качестве десятичного разделителя точку или запятую. Эта информация используется в функции **strfmon**[9]**(3)**.

LC_MESSAGES* — эта категория изменяет язык отображаемых сообщений и указывает, как должны выглядеть положительный и отрицательный ответы.

Библиотека GNU C содержит функции **gettext(3)**, **ngettext(3)** и **rpmatch(3)** для более удобного использования этой информации. Семейство функций GNU **gettext** так же учитывает переменную окружения **LANGUAGE** (содержащую через двоеточие список локалей), если указана допустимая локаль, отличная от **"C"**. Данная категория также влияет на поведение **catopen(3)**.

---

6 Преобразует текст из одной кодировки в другую

7 mblen() – определяет количество байтов в последующем многобайтовом символе

8 wctomb() – преобразует широкий символ в многобайтовую последовательность

9 strfmon() – преобразует денежное значение в строку

LC_NUMERIC — эта категория определяет правила форматирования, используемые для не денежных значений, например, разделительный символ тысяч и дробной части (точка в англоязычных странах, и запятая во многих других).

Она влияет на поведение функций **printf(3)**, **scanf(3)** и **strtod(3)**.

Эта информация может быть также прочитана при помощи функции **localeconv(3)**.

LC_TIME — эта категория управляет форматированием значений даты и времени. Например, большая часть Европы использует 24-часовой формат, тогда как в США используют 12-часовой-.Значение этой категории влияет на поведение функции **strftime**[10]**(3)** и **strptime**[11]**(3)**.

LC_ALL — всё вышеперечисленное.

---

10 strftime() — форматирование даты и времени

11 strptime() — конвертирует строчное представление времени в представление структуры времени tm

setlocale() — задаёт текущую локаль

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Функция **setlocale()** используется для назначения или запроса текущей локали для программы.

Если **locale** не равно **NULL**, то текущая локаль программы изменяется согласно переданным аргументам.

Аргументом **category** определяется какую часть текущей локали программы нужно изменить.

Если второй аргумент функции **setlocale(3)** равен пустой строке **""**, то локаль по умолчанию будет определяться следующим образом:

1. Если существует непустая переменная окружения **LC_ALL**, то используется её значение.
2. Если существует переменная окружения с именем одной из вышеописанных категорий и она непустая, то её значение присваивается этой категории.
3. Если существует непустая переменная окружения **LANG**, то используется её значение.

Информация о локальном форматировании чисел доступна в структуре **struct lconv**, возвращаемой функцией **localeconv(3)**, которая содержит элементы, связанные с форматированием числовых значений.

В **setlocale(3)** используется переменная окружения **LOCPATH** и если она установлена, влияет на все непривилегированные локализованные программы.

Эта переменная представляет список путей, разделённых двоеточием («**:**»), который должен использоваться при поиске данных, ассоциированных с локалью.

Скомпилированные файлы данных локали ищутся в подкаталогах, которые в данный момент зависят от используемой локали. Например, если для категории используется **en_GB.UTF-8**, то будут просмотрены следующие подкаталоги в таком порядке:

en_GB.UTF-8, en_GB.utf8, en_GB, en.UTF-8, en.utf8 и en

/usr/lib/locale/locale-archive — обычный путь по умолчанию для расположения архива локалей.

/usr/lib/locale — обычный путь по умолчанию для скомпонованных файлов отдельных локалей.

Если **`locale`** не равно **`NULL`**, то текущая локаль программы изменяется согласно переданным аргументам.

Аргумент **`category`** определяет, какую часть текущей локали программы нужно изменить.

Категория	**Назначение**
`LC_ALL`	локаль целиком
`LC_COLLATE`	сортировка строк
`LC_CTYPE`	классы символов
`LC_MESSAGES`	локализированные сообщения на родном языке
`LC_MONETARY`	форматирование значений денежных единиц
`LC_NUMERIC`	форматирование не денежных числовых значений
`LC_TIME`	форматирование значений дат и времени

Список поддерживаемых категорий можно получить с помощью утилиты оболочки locale

```
$ locale
LANG=ru_RU.utf8
LC_CTYPE="ru_RU.utf8"
LC_NUMERIC="ru_RU.utf8"
LC_TIME="ru_RU.utf8"
LC_COLLATE="ru_RU.utf8"
LC_MONETARY="ru_RU.utf8"
LC_MESSAGES="ru_RU.utf8"
LC_PAPER="ru_RU.utf8"
LC_NAME="ru_RU.utf8"
LC_ADDRESS="ru_RU.utf8"
LC_TELEPHONE="ru_RU.utf8"
LC_MEASUREMENT="ru_RU.utf8"
LC_IDENTIFICATION="ru_RU.utf8"
LC_ALL=
```

Аргумент **locale** — это указатель на строку символов, содержащую требуемую настройку для **category**.

Эта строка может быть константой «**C**» или «**ru_RU**», или строкой со скрытым форматом, которую возвращает другой вызов **setlocale()**.

Если **locale** — пустая строка **""**, то любая часть локали, которую требуется изменить, будет задана исходя из переменных окружения.

Как это происходит — зависит от реализации.

В glibc, во-первых (независимо от **category**), просматривается переменная окружения **LC_ALL**, затем переменная окружения с именем как у категории (смотрите таблицу выше), и в конце учитывается переменная окружения **LANG**.

Используется первая найденная переменная окружения.

Если её значение некорректно определяет локаль, то локаль не изменяется и **setlocale()** возвращает NULL.

Локали **"C"** или **"POSIX"** являются переносимыми локалями — они существуют во всех соответствующих системах.

Имя локали обычно имеет вид:

```
language[_terrirory][chartable][@modifier]
```

language (язык) — код языка согласно ISO 639;
terrirory (территория) — код страны согласно ISO 3166;
chartable (таблица символов) — набор символов или кодировка типа ISO-8859-1 или UTF-8.

Список поддерживаемых локалей можно получить по команде **locale -a**.

```
$ locale -a
C
C.utf8
...
en_US
...
POSIX
ru_RU
ru_RU.koi8r
ru_RU.utf8
ru_UA
ru_UA.utf8
```

Если **locale** равно **NULL**, то только возвращается текущая локаль и ничего не меняется.

При запуске **main()** по умолчанию выбирается переносимая локаль **"C"**.

Программу можно сделать переносимой для всех локалей, вызвав

```
setlocale(LC_ALL, "");
```

При успешном выполнении **setlocale()** возвращает строку со скрытым форматом, которая соответствует набору локали.

Эта строка может находиться в статической памяти.

Строка возвращается таким образом, что последующий вызов с этой строкой и связанной с ней категорией, восстанавливают эту часть локали процесса. Если вызов не может быть выполнен, то возвращается значение **NULL**.

localeconv() — получить информацию о форматировании числовых данных

```
#include <locale.h>
struct lconv *localeconv(void);
```

Функция **localeconv()** возвращает указатель на структуру **struct lconv** для текущей локали.

Эта структура определена в файле заголовков **locale.h (locale 7)** и содержит все значения, связанные с категориями локали **LC_NUMERIC** и **LC_MONETARY**.

Программы также могут использовать функции **printf(3)** и **strfmon(3)**, поведение которых зависит от того, какая локаль сейчас используется.

Функция **localeconv()** возвращает указатель на структуру **struct lconv**.

Под эту структуру может (в glibc так и есть) быть статически выделена память, и она может быть перезаписана следующими вызовами.

Согласно POSIX, вызывающий не должен изменять содержимое структуры.

Функция **localeconv()** всегда завершается без ошибок.

Члены структуры с типом **char *** являются указателями на строки, любая из которых (кроме **decimal_point**) может указывать на **""** — это означает, что значение недоступно в текущей локали или имеет нулевую длину.

Помимо **grouping** и **mon_grouping**, строки должны начинаться и заканчиваться в начальном состоянии сдвига.

Элементы с типом **char** являются неотрицательными числами, любое из которых может быть **CHAR_MAX** — это означает, что значение недоступно в текущей локали.

```
$ cat /usr/include/locale.h
#ifndef  _LOCALE_H
#define  _LOCALE_H1

#include <features.h>

#define __need_NULL
#include <stddef.h>
#include <bits/locale.h>

__BEGIN_DECLS

/* These are the possibilities for the first argument to setlocale.
   The code assumes that the lowest LC_* symbol has the value zero.  */
#define LC_CTYPE          __LC_CTYPE
#define LC_NUMERIC        __LC_NUMERIC
#define LC_TIME           __LC_TIME
#define LC_COLLATE        __LC_COLLATE
#define LC_MONETARY       __LC_MONETARY
#define LC_MESSAGES       __LC_MESSAGES
#define LC_ALL            __LC_ALL
#define LC_PAPER          __LC_PAPER
#define LC_NAME           __LC_NAME
#define LC_ADDRESS        __LC_ADDRESS
#define LC_TELEPHONE      __LC_TELEPHONE
#define LC_MEASUREMENT    __LC_MEASUREMENT
#define LC_IDENTIFICATION __LC_IDENTIFICATION
```

```
/* Structure giving information about numeric and monetary notation.  */
struct lconv { /* Numeric (non-monetary) information.  */
  char *decimal_point;     /* Decimal point character.  */
  char *thousands_sep;     /* Thousands separator.  */
...

  /* Monetary information.  */
  /* First three chars are a currency symbol from ISO 4217. */
  /* Fourth char is the separator.  Fifth char is '\0'.  */
  char *int_curr_symbol;
  char *currency_symbol;   /* Local currency symbol.  */
  char *mon_decimal_point; /* Decimal point character.  */
  char *mon_thousands_sep; /* Thousands separator.  */
  char *mon_grouping;      /* Like `grouping' element (above).  */
  char *positive_sign;     /* Sign for positive values.  */
  char *negative_sign;     /* Sign for negative values.  */
  ...
};

/* Set and/or return the current locale.  */
extern char *setlocale (int __category, const char *__locale) __THROW;
/* Return the numeric/monetary information for the current locale.  */
extern struct lconv *localeconv (void) __THROW;
...
__END_DECLS
#endif /* locale.h  */
```

Библиотека. Базовые утилиты

Утилиты поиска и сортировки

qsort() – сортировка элементов массива.
bsearch() – двоичный поиск в массиве, отсортированном по ключу;

```
#include <stdlib.h>

void    qsort(void *base,       // начальный элемент массива
              size_t nmemb,     // количество объектов (элементов массива)
              size_t size,      // размер объекта (элемента массива)
              int (*compar)(const void *, const void *));

void *bsearch(const void *key,  // ключ поиска
              const void *base, // начальный элемент массива
              size_t nmemb,     // количество объектов (элементов массива)
              size_t size,      // размер объекта (элемента массива)
              int (*compar)(const void *, const void *));
```

Функция qsort

```
#include <stdlib.h>

void qsort(void *base,   // начальный элемент массива
           size_t nmemb, // количество элементов массива
           size_t size,  // размер объекта (размер элемента массива)
           int (*compar)(const void *, const void *));
```

Функция **qsort** сортирует массив **nmemb** объектов, на начальный элемент которых указывает **base**. Размер каждого объекта определяется **size**.

Содержимое массива сортируется в порядке возрастания в соответствии с функцией сравнения, на которую указывает **compar**, и которая вызывается с двумя аргументами, указывающими на сравниваемые объекты.

Функция должна возвращать целое число меньше, равно или больше нуля, если первый аргумент считается соответственно меньше, равен или больше второго.

Если два элемента сравниваются как равные, их порядок в результирующем отсортированном массиве не определен.

Функция **qsort** не возвращает значения.

Функция bsearch

```
#include <stdlib.h>

void *bsearch(const void *key,  // ключ поиска
              const void *base, // начальный элемент массива
              size_t nmemb,     // количество объектов (размер массива)
              size_t size,      // размер элемента
              int (*compar)(const void *, const void *));
```

Функция **bsearch** ищет в массиве из **nmemb** объектов, на начальный элемент которых указывает **base**, элемент, соответствующий объекту, на который указывает **key**. Размер каждого элемента массива определяется **size**.

Функция сравнения, на которую указывает **compar**, вызывается с двумя аргументами, которые указывают на объект **key** и на элемент массива.

Функция должна возвращать целое число меньше, равно или больше нуля, если ключевой объект считается, соответственно, меньше, соответствует или больше, чем элемент массива.

Массив должен быть упорядочен в ключе. На практике перед вызовом **bsearch** весь массив сортируется в соответствии с функцией сравнения.

Функция **bsearch** возвращает указатель на соответствующий элемент массива или, если совпадений не найдено, нулевой указатель.

Требования к compar

compar — указатель на функцию, которая сравнивает два элемента. Эта функция многократно вызывается функцией **bsearch** для сравнения **key** с отдельными элементами в **base**, а также функцией **qsort** для сравнения элементов масива.

Функция **compar** должен следовать следующему прототипу:

```
int compar(const void* p1, const void* p2);
```

Принимает два указателя в качестве аргументов.

Для функции **bsearch** первый всегда является указателем на ключ, а второй указывает на элемент массива (оба преобразуются в **const void ***).

Для функции **qsort**, оба указывают на элементы массива (преобразуются в **const void ***).

Функция должна возвращать (устойчивым и транзитивным образом) следующие значения:

<0 Элемент, на который указывает **pkey**, идет перед элементом, на который указывает **pelem**;

0 Элемент, на который указывает **pkey**, эквивалентен элементу, на который указывает **pelem**;

>0 Элемент, на который указывает **pkey**, идет после элемента, на который указывает **pelem**.

Для типов, которые можно сравнивать с помощью обычных реляционных операторов, общая функция сравнения может выглядеть следующим образом:

```
int compare_my_type(const void *a, const void *b) {
    if (*(my_type*)a <  *(my_type*)b ) return -1;
    if (*(my_type*)a == *(my_type*)b ) return 0;
    if (*(my_type*)a >  *(my_type*)b ) return 1;
}
```

Пример

```
/* bsearch example */
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* qsort, bsearch, NULL */

int compareints(const void *a, const void *b) {
  return ( *(int *)a - *(int *)b );
}

int values[] = { 50, 20, 60, 40, 10, 30 };

int main () {
    int * pItem;
    int key = 40;
    qsort(values, 6, sizeof (int), compareints);
    pItem = (int*)bsearch(&key, values, 6, sizeof(int), compareints);
    if (pItem != NULL)
        printf ("%d присутствует в массиве.\n", *pItem);
    else
        printf ("%d отсутствует в массиве.\n", key);
    return 0;
}
```

В этом примере **compareints** сравнивает значения, на которые указывают два параметра, как значения типа **int** и возвращает результат их вычитания, результат которого равен 0, если они равны, положителен, если значение, на которое указывает **a**, больше, чем значение, на который указывает **b**, или отрицателен, если значение, на которое указывает **b**, больше, чем **a**.

В основной функции целевой массив сортируется с помощью **qsort** перед вызовом **bsearch** для поиска значения.

Вывод:

40 присутствует в массиве.

Для C-строк в качестве аргумента сравнения для **bsearch** может напрямую использоваться **strcmp()**:

```
/* bsearch example with strings */
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* qsort, bsearch, NULL */
#include <string.h>     /* strcmp */

char strvalues[][20] = {"some","example","strings","here"};

int main () {

   char * pItem;
   char key[20] = "example";

   /* sort elements in array: */
   qsort(strvalues, 4, 20, (int(*)(const void*,const void*))strcmp);

   /* search for the key: */
   pItem = (char*)bsearch(key, strvalues, 4, 20,
                          (int(*)(const void*,const void*))strcmp);
   if (pItem!=NULL)
      printf ("%s is in the array.\n",pItem);
   else
      printf ("%s is not in the array.\n",key);
   return 0;
}
```

Функции целочисленной арифметики

Функции abs, labs и llabs

```
#include <stdlib.h>

int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
```

Функции **abs**, **labs** и **llabs** вычисляют абсолютное значение целого числа **j**.
Если результат не может быть представлен в типе, поведение не определено.

Возврат
Функции **abs**, **labs** и **llabs** возвращают абсолютное значение аргумента.

Функции div, ldiv и lldiv

```
#include <stdlib.h>

div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
```

Функции **div**, **ldiv** и **lldiv** вычисляют результат целочисленного деления делимого **numer** на делитель **denom** и остаток от этого деления за одно действие.

Возврат

Функции **div**, **ldiv** и **lldiv** возвращают структуру типа **div_t**, **ldiv_t** и **lldiv_t**, соответственно, содержащую как частное, так и остаток.

Структуры содержат (в любом порядке) члены **quot** (частное) и **rem** (остаток), каждый из которых имеет тот же тип, что и аргументы **numer** и **denom**. Если какая-либо часть результата не может быть представлена в типе, поведение не определено.

```
typedef struct { // C98, C++11
    long long int quot;         /* Quotient.  */
    long long int rem;          /* Remainder.  */
} lldiv_t;
```

Пример

```
/*
 * lldiv example
 */
#include <stdio.h>      // printf
#include <stdlib.h>     // lldiv, lldiv_t

int main () {

  lldiv_t res = lldiv(31558149LL, 3600LL);

  printf("Оборот Земли по орбите: %lld часов and %lld секунд.\n",
         res.quot,
         res.rem);
  return 0;
}
```

Вывод

```
Оборот Земли по орбите: 8766 часов and 549 секунд.
```

Библиотека. Широкие символы

<cwchar> (wchar.h) – этот заголовочный файл определяет несколько функций для работы со строками широких символов и вводит несколько типов.

Типы

mbstate_t – состояние многобайтового преобразования
size_t – беззнаковый целочисленный тип
struct tm – структура времени
wchar_t – широкий символ
wint_t – широкий целочисленный тип

mbstate_t - состояние многобайтового преобразования

Это тип, содержащий информацию, необходимую для сохранения состояния при преобразованиях между последовательностями многобайтовых символов и широких символов (в любом направлении).

Кодировка многобайтовой последовательности может иметь разные состояния сдвига, которые меняют способ интерпретации следующего байтового символа. Значения типа **mbstate_t** могут сохранять эти состояния между вызовами функций, так что перевод последовательности может безопасно выполняться при осуществлении нескольких вызовов.

Все допустимые многобайтовые последовательности должны начинаться (и заканчиваться) в одном и том же состоянии (называемом его начальным состоянием).

Объект **mbstate_t** с нулевым значением всегда описывает начальное состояние преобразования, хотя другие значения также могут представлять такое состояние (в зависимости от конкретной реализации библиотеки).

Объект типа **mbstate_t (mbs)** можно установить в исходное состояние, вызвав:

```
memset(&mbs, 0, sizeof(mbs));  // mbs is now a zero-valued object
```

Для проверки конкретного состояния два значения **mbstate_t** не должны сравниваться друг с другом — статус начального состояния объекта с типом **mbstate_t** можно проверить с помощью функции **mbsinit()**.

wchar_t — тип широкого символа

Тип, диапазон значений которого может представлять различные коды для всех членов самого большого расширенного набора символов, указанного среди поддерживаемых локалей.

В C++ **wchar_t** — это отдельный фундаментальный тип (и поэтому он не определен ни в **<cwchar>**, ни в каком-либо другом заголовке).

В C это **typedef** целочисленного типа достаточного размера.

wint_t — широкий целочисленный тип

Это **typedef** типа, способного представить любое значение типа **wchar_t**, которое является членом расширенного набора символов, а также дополнительное значение **WEOF** (не является частью этого набора).

Этот тип может быть либо псевдонимом **wchar_t**, либо псевдонимом целочисленного типа, в зависимости от конкретной реализации библиотеки. Но он должен оставаться неизменным при продвижении аргументов по умолчанию из значений типа **wchar_t**.

Многобайтовые (мультибайтовые) символы

mblen — Получить длину многобайтового символа (функция)
mbtowc — Преобразование многобайтовой последовательности в широкий символ (функция)

mblen — получить длину многобайтового символа

```
#include <stdlib.h>
int mblen(const char *pmb, size_t max);
```

Возвращает размер многобайтового символа, указанного **pmb**, проверяя не более **max** байт.

mblen имеет собственное внутреннее состояние сдвига, которое при необходимости изменяется только при вызове этой функции.

Поведение этой функции зависит от категории **LC_CTYPE** выбранной локали.

pmb — указатель на первый байт многобайтового символа.

В качестве альтернативы функция может быть вызвана с нулевым указателем, и в этом случае функция сбрасывает свое внутреннее состояние сдвига на начальное значение и возвращает, используют ли многобайтовые символы кодировку, зависящую от состояния.

max — максимальное количество байтов **pmb**, которое следует учитывать для многобайтового символа.

В любом случае проверяется не более чем **MB_CUR_MAX** символов.

size_t — это целочисленный тип без знака.

Возвращаемое значение

Если аргумент, переданный как **pmb**, не является нулевым указателем, возвращается размер символа, на который указывает **pmb**, в байтах, если он образует допустимый многобайтовый символ, но не завершающий нулевой символ.

Если это завершающий нулевой символ, функция возвращает ноль, а если байты **pmb** не образуют допустимый многобайтовый символ, возвращается -1.

Если аргумент, переданный в качестве **pmb**, является нулевым указателем, функция возвращает ненулевое значение, если кодировки многобайтовых символов зависят от состояния, и ноль в противном случае.

Пример

В примере используется строка с использованием локали "C".

```
#include <stdio.h>      // printf
#include <stdlib.h>     // mblen, mbtowc, wchar_t(C)

// ------------------------------------------
// печатает многобайтовую строку посимвольно.
// ------------------------------------------
void printbuffer(const char *pt, size_t max) {

  int     length;
  wchar_t dest;

  mblen(NULL, 0);         // reset mblen
  mbtowc(NULL, NULL, 0);  // reset mbtowc

  while(max > 0) {
      length = mblen(pt, max);
      if (length < 1) break;     // мусор в строке — это не символ
      mbtowc(&dest, pt, length);
      printf("[%lc]", dest);
      pt += length; max -= length;
  }
}
```

```
int main( ) {

  const char str[] = "test string";

  printbuffer(str, sizeof(str));
  return 0;
}
```

Вывод:

```
[t][e][s][t][ ][s][t][r][i][n][g]
```

Гонки данных

Функция обращается к массиву, указанному **pmb**.

Функция также обращается к объекту внутреннего состояния и изменяет его, что может вызвать гонки данных при одновременных вызовах этой функции.

Есть альтернатива, которая может использовать объект внешнего состояния. Это функция **mbrlen**.

Одновременное изменение настроек локали также может привести к гонке данных.

Если **pmb** не является ни пустым указателем, ни указателем на достаточно длинный массив, это вызовет неопределенное поведение.

mbtowc — преобразование многобайтовой последовательности в широкий символ

```
int mbtowc(wchar_t *pwc,    // широкий результат
           const char *pmb, // многобайтовый источник
           size_t max);     // максимальное кол-во байт в многобайтовом символе
```

Многобайтовый символ, на который указывает **pmb**, преобразуется в значение типа **wchar_t** и сохраняется в месте, указанном **pwc**.

Функция возвращает длину многобайтового символа в байтах.

mbtowc имеет собственное внутреннее состояние сдвига, которое изменяется по мере необходимости только при вызове этой функции.

Вызов функции с нулевым указателем в качестве **pmb** сбрасывает состояние (и возвращает, зависят ли многобайтовые символы от состояния).

Поведение этой функции зависит от категории **LC_CTYPE** выбранной С-локали.

Параметры

pwc — указатель на объект типа **wchar_t**.

В качестве альтернативы, этот аргумент может быть нулевым указателем, и в этом случае функция не сохраняет перевод **wchar_t**, но по-прежнему возвращает длину в байтах многобайтового символа.

pmb — указатель на первый байт многобайтового символа.

В качестве альтернативы, этот аргумент может быть нулевым указателем, и в этом случае функция сбрасывает свое внутреннее состояние сдвига до начального значения и возвращает, имеют ли многобайтовые символы кодировку, зависящую от состояния.

max — максимальное количество байтов **pmb**, которое следует учитывать для многобайтового символа.

В любом случае проверяется не более чем **MB_CUR_MAX** символов.

Возвращаемое значение

Если аргумент, переданный как **pmb**, не является нулевым указателем, возвращается размер символа, на который указывает **pmb**, в байтах, если он образует допустимый многобайтовый символ, но не завершающий нулевой символ.

Если это завершающий нулевой символ, функция возвращает ноль, а если байты **pmb** не образуют допустимый многобайтовый символ, возвращается -1.

Если аргумент, переданный в качестве **pmb**, является нулевым указателем, функция возвращает ненулевое значение, если кодировки многобайтовых символов зависят от состояния, и ноль в противном случае.

Пример

В примере используется строка с использованием локали "C".

```
// mbtowc example
#include <stdio.h>      // printf */
#include <stdlib.h>     // mbtowc, wchar_t(C)

// печатает многобайтовую строку посимвольно.
void printbuffer (const char* pt, size_t max) {

  int length;
  wchar_t dest;

  mbtowc (NULL, NULL, 0);  // reset mbtowc

  while (max>0) {
    length = mbtowc(&dest, pt, max);
    if (length<1) break;
    printf("[%lc]", dest);
    pt += length;
    max -= length;
  }
}
```

```
int main( ) {

  const char str[] = "mbtowc example";

  printbuffer(str, sizeof(str));

  return 0;
}
```

Вывод

```
[m][b][t][o][w][c][ ][e][x][a][m][p][l][e]
```

Гонки данных

Функция обращается к массиву, указанному **pmb**, и изменяет объект, указанный **pwc** (если не **null**).

Функция также обращается к объекту внутреннего состояния и изменяет его, что может вызвать гонки данных при одновременных вызовах этой функции. Есть альтернатива, которая может использовать объект внешнего состояния. Это функция **mbrtowc**.

Одновременное изменение настроек локали также может привести к гонке данных.

Если **pmb** не является ни пустым указателем, ни указателем на достаточно длинный массив, это вызовет неопределенное поведение.

wctomb — преобразование широкого символа в многобайтовую последовательность

```
int wctomb(char* pmb,    // многобайтовый результат
           wchar_t wc);  // широкий источник
```

Широкий символ **wc** преобразуется в его многобайтовый эквивалент и сохраняется в массиве, на который указывает **pmb**.

Функция возвращает длину в байтах эквивалентной многобайтовой последовательности, указанной **pmb** после вызова.

wctomb имеет собственное внутреннее состояние сдвига, которое изменяется по мере необходимости только вызовами этой функции. Вызов функции с нулевым указателем в качестве **pmb** сбрасывает состояние (и возвращает, зависят ли многобайтовые последовательности от состояния).

Поведение этой функции зависит от категории **LC_CTYPE** выбранной C-локали.

Гонки данных

Функция изменяет массив, на который указывает **pmb**.

Функция также обращается к объекту внутреннего состояния и изменяет его, что может вызвать гонки данных при одновременных вызовах этой функции (см. **wcrtomb** d качестве альтернативы, которая может использовать объект внешнего состояния).

Одновременное изменение настроек локали также может привести к гонке данных.

Параметры

pmb — указатель на массив, достаточно большой для хранения многобайтовой последовательности. Максимальная длина многобайтовой последовательности для символа в текущей локали составляет **MB_CUR_MAX** байт.

В качестве альтернативы функция может быть вызвана с нулевым указателем, и в этом случае функция сбрасывает свое внутреннее состояние сдвига на начальное значение и возвращает, используют ли многобайтовые последовательности кодирование, зависящее от состояния.

wc — широкий символ типа **wchar_t**.

Возвращаемое значение

Если аргумент, переданный как **pmb**, не является нулевым указателем, возвращается размер в байтах символа, записанного в **pmb**.

Если символьного соответствия нет, возвращается -1.

Если аргумент, переданный как **pmb**, является нулевым указателем, функция возвращает ненулевое значение, если кодировки многобайтовых символов зависят от состояния, и ноль в противном случае.

Пример

В примере печатаются многобайтовые символы, в которые с использованием выбранной локали преобразуется строка широких символов (в данном случае "C" по умолчанию).

```
#include <stdio.h>      // printf
#include <stdlib.h>     // wctomb, wchar_t(C)

int main() {

  const wchar_t  str[] = L"wctomb example";
  const wchar_t *pt    = str;
  char           buffer[MB_CUR_MAX];
  int            i,length;

  while (*pt) {
    length = wctomb(buffer, *pt++);
    if (length < 1) break;          // Ошибка
    for (i = 0; i < length; ++i) {
      printf ("[%c]",buffer[i]);
    }
  }
  return 0;
}
```

Вывод:

```
[w][c][t][o][m][b][ ][e][x][a][m][p][l][e]
```

Многобайтовые (мультибайтовые) строки

`mbstowcs` — Преобразование многобайтовой строки в строку расширенных символов
`wcstombs` — Преобразование строки расширенных символов в многобайтовую строку

mbstowcs — преобразование многобайтовой строки в строку расширенных символов

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *dest,    // куда
                const char *src,  // откуда
                size_t max);      // максимальное кол-во широких символов
```

Преобразует многобайтовую последовательность, указанную **src**, в эквивалентную последовательность широких символов (которые сохраняются в массиве, указанном **dest**), до тех пор, пока не будут преобразовано **max** широких символов или пока не встретится нулевой символ в многобайтовой последовательности **src** (который также преобразуется и сохраняется, но не учитывается в длине, возвращаемой функцией).

Если **max** символов преобразовано успешно, результирующая строка, хранящаяся в **dest**, не будет завершена нулем.

Поведение этой функции зависит от категории **LC_CTYPE** выбранной С-локали.

Параметры

dest — указатель на массив элементов **wchar_t**, достаточно длинный, чтобы содержать результирующую последовательность (не более, чем **max** широких символов).

src — С-строка с интерпретируемыми многобайтовыми символами.

Многобайтовая последовательность должна начинаться в состоянии начального сдвига.

max — максимальное количество символов **wchar_t** для записи в **dest**.

Возвращаемое значение

Число широких символов, записанных в **dest**, не включая конечный нулевой символ.

Если обнаружен недопустимый многобайтовый символ, возвращается значение **(size_t)-1**.

Следует обратить внимание, что **size_t** - это целочисленный тип без знака, и поэтому ни одно из возможных возвращаемых значений не будет меньше нуля.

Гонки данных

Функция обращается к массиву, указанному **src**, и изменяет массив, указанный **dest**. Функция также может обращаться к объекту внутреннего состояния и изменять его, что может вызывать гонки данных при одновременных вызовах этой функции, если реализация использует статический объект (см. **mbsrtowcs** для альтернативы, которая может использовать объект внешнего состояния).

Одновременное изменение настроек локали также может привести к гонке данных.

wcstombs — преобразование строки расширенных символов в многобайтовую строку

```
#include <stdlib.h>
size_t wcstombs(char *dest,          // куда писать байты
                const wchar_t *src, // откуда рать широкие символы
                size_t max);        // максимальное кол-во байт для преобразов.
```

Преобразует широкие символы из последовательности, указанной **src**, в многобайтовую эквивалентную последовательность (которая сохраняется в массиве, указанном **dest**), до тех пор, пока не будет преобразовано **max** байтов или пока широкие символы не преобразуются в нулевой символ.

Если **max** байтов успешно переведено, результирующая строка, хранящаяся в **dest**, не будет завершена нулем.

Результирующая многобайтовая последовательность начинается в начальном состоянии сдвига (если есть).

Поведение этой функции зависит от категории **LC_CTYPE** выбранной С-локали.

Параметры

dest — указатель на массив элементов **char**, достаточно длинный, чтобы содержать результирующую последовательность (не более, чем **max** байтов).

src — строка широких символов для преобразования.

max — максимальное количество байтов для записи в **dest**.

Возвращаемое значение

Количество байтов, записанных в **dest**, не включает конечный нулевой символ.

Если встречается широкий символ, не соответствующий допустимому многобайтовому символу, возвращается значение **(size_t)-1**.

Следует обратить внимание, что **size_t** - это целочисленный тип без знака, и поэтому ни одно из возможных возвращаемых значений не будет меньше нуля.

Пример

```
/* wcstombs example */
#include <stdio.h>      /* printf */
#include <stdlib.h>     /* wcstombs, wchar_t(C) */

int main() {
  const wchar_t str[] = L"wcstombs example";
  char buffer[32];

  printf("wchar_t string: %ls \n",str);

  int ret = wcstombs(buffer, str, sizeof(buffer));
  if (ret == 32) {
    buffer[31]='\0';
  }
  if (ret) {
    printf("multibyte string: %s \n",buffer);
  }
  return 0;
}
----------------------------------------
wchar_t string: wcstombs example
multibyte string:  wcstombs example
```

Гонки данных

Функция обращается к массиву, указанному **`src`**, и изменяет массив, указанный **`dest`**. Функция также может обращаться к объекту внутреннего состояния и изменять его, что может вызывать гонки данных при одновременных вызовах этой функции, если реализация использует статический объект (см. **`wcsrtombs`** для альтернативы, которая может использовать объект внешнего состояния).

Одновременное изменение настроек локали также может привести к гонке данных.