17
Обзор функций

В этой главе перечислены функции стандартной библиотеки в соответствии с областями их применения, описаны их особенности и взаимоотношения одна с другой. Эта глава может помочь вам найти нужную функцию для своих целей во время работы над программой.

Отдельные функции подробно описаны в главе 18, “Функции стандартной библиотеки”, где они приведены в алфавитном порядке и с примерами.

Альтернативные функции с проверкой выхода за границы в C11, именуемые также “безопасными” функциями, перечислены в табл. 17.1 и 17.2. Имена этих функций заканчиваются суффиксом `_s`, как, например, `scanf_s()`. Обратите внимание, что от реализации C не требуется поддержка безопасных функций. Дополнительную информацию об использовании безопасных функций вы найдете в разделе “Функции с проверкой выхода за границы” главы 16, “Стандартные заголовочные файлы”.

Ввод и вывод

Эта тема детально обсуждалась в главе 13, “Ввод-вывод”, в который содержатся разделы, посвященные потокам ввода-вывода, последовательному и произвольному доступу к файлам, форматированному вводу-выводу и обработке ошибок. Поэтому здесь достаточно просто табличного списка функций ввода-вывода. В табл. 17.1 приведены общие функции для работы с файлами, объявленные в заголовочном файле `stdio.h`.

Таблица 17.1. Функции общего назначения для работы с файлами

Цель	Функции
Переименование, удаление файла	`rename()`, `remove()`
Создание или открытие файла	`fopen()`, `freopen()`, `tmpfile()` `fopen_s()`, `freopen_s()`, `tmpfile_s()`
Закрытие файла	`fclose()`
Генерация уникального имени файла	`tmpnam()`, `tmpnam_s()`
Запрос или сброс флагов доступа к файлу	`feof()`, `ferror()`, `clearerr()`
Запрос текущей позиции в файле	`ftell()`, `fgetpos()`
Изменение текущей позиции в файле	`rewind()`, `fseek()`, `fsetpos()`
Сброс буфера с содержимым в файл	`fflush()`
Управление буферизацией файла	`setbuf()`, `setvbuf()`

Имеется два комплекта функций для ввода-вывода символов и строк: байтные и широкосимвольные (дополнительная информация имеется в разделе "Байтные и широкие потоки" главы 13, "Ввод-вывод"). Широкосимвольные функции работают с типом `wchar_t` и объявлены в заголовочном файле `wchar.h`. В табл. 17.2 содержатся оба комплекта.

Таблица 17.2. Функции файлового ввода-вывода

Цель	Функции в `stdio.h`	Функции в `wchar.h`
Получение/установка направленности потока		`fwide()`
Запись символов	`fputc()`, `putc()`, `putchar()`	`fputwc()`, `putwc()`, `putwchar()`
Чтение символов	`fgetc()`, `getc()`, `getchar()`	`fgetwc()`, `getwc()`, `getwchar()`
Возврат в поток считанного символа	`ungetc()`	`ungetwc()`
Запись строк	`fputs()`, `puts()`	`fputws()`
Чтение строк	`fgets()`, `gets()`, `gets_s()`	`fgetws()`
Запись блоков	`fwrite()`	
Чтение блоков	`fread()`	
Запись отформатированных строк	`printf()`, `vprintf()` `fprintf()`, `vfprintf()` `sprintf()`, `vsprintf()` `snprintf()`, `vsnprintf()`	`wprintf()`, `vwprintf()` `fwprintf()`, `vfwprintf()` `swprintf()`, `vswprintf()`
Чтение отформатированных строк	`scanf()`, `vscanf()` `fscanf()`, `vfscanf()` `sscanf()`, `vsscanf()`	`wscanf()`, `vwscanf()` `fwscanf()`, `vfwscanf()` `swscanf()`, `vswscanf()`

Для каждой функции семейств `printf` и `scanf` имеется безопасная альтернативная функция, имя которой заканчивается суффиксом `_s`.

Математические функции

Стандартная библиотека предоставляет множество математических функций. Большинство из них предназначены для работы с действительными или комплексными числами с плавающей точкой. Однако имеется также ряд функций для работы с целочисленными типами, таких как функции для генерации случайных чисел.

Функции для преобразования строк с числовыми значениями в арифметические типы перечислены в разделе "Работа со строками" данной главы. Прочие математические функции описаны в следующих подразделах.

Математические функции для целочисленных типов

Математические функции для целочисленных типов объявлены в заголовочном файле `stdlib.h`. Две из этих функций, `abs()` и `div()`, объявлены в трех вариантах для работы с тремя знаковыми целочисленными типами `int`, `long` и `long long`. Как показано в табл. 17.3, функции для типа `long` имеют имена, начинающиеся с буквы `l`; для типа `long long` — с `ll`. Кроме того, в заголовочном файле `inttypes.h` объявлены версии функций для типа `intmax_t` с именами, начинающимися с `imax`.

Таблица 17.3. Целочисленные арифметические функции

Назначение	Функции, объявленные в `stdlib.h`	Функции, объявленные в `stdint.h`
Абсолютное значение	`abs()`, `labs()`, `llabs()`	`imaxabs()`
Деление	`div()`, `ldiv()`, `lldiv()`	`imaxdiv()`
Случайные числа	`rand()`, `srand()`	

Функции для работы с числами с плавающей точкой

Функции для работы с действительными типами с плавающей точкой объявлены в заголовочном файле `math.h`, а функции для работы с комплексными числами с плавающей точкой — в заголовочном файле `complex.h`. В табл. 17.4 перечислены функции, доступные как для действительных, так и для комплексных типов с плавающей точкой. Комплексные версии функций имеют имена, начинающиеся с буквы `c`. В табл. 17.5 перечислены функции, определенные только для действительных типов, а в табл. 17.6 — функции для работы только с комплексными типами.

Для удобочитаемости в табл. 17.4–17.6 приведены имена функций только для типов `double` и `double _Complex`. Каждая из этих функций имеется также в вариантах для типов `float` (или `float _Complex`) и `long double` (или `long double _Complex`). Имена этих вариантов заканчиваются на `f` для `float` и на `l` для `long double`. Например, функции `sin()` и `csin()`, приведенные в табл. 17.4, имеются также в вариантах `sinf()`, `sinl()`, `csinf()` и `csinl()` (однако читайте также раздел "Макросы, не зависящие от типа" далее в этой главе).

Таблица 17.4. Функции для действительных и комплексных типов с плавающей точкой

Математические функции	Функции C из `math.h`	Функции C из `complex.h`
Тригонометрия	`sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`	`csin()`, `ccos()`, `ctan()`, `casin()`, `cacos()`, `catan()`
Гиперболическая тригонометрия	`sinh()`, `cosh()`, `tanh()`, `asinh()`, `acosh()`, `atanh()`	`casinh()`, `cacosh()`, `catanh()`, `csinh()`, `ccosh()`, `ctanh()`
Экспоненциальная функция	`exp()`	`cexp()`
Натуральный логарифм	`log()`	`clog()`
Возведение в степень, квадратный корень	`pow()`, `sqrt()`	`cpow()`, `csqrt()`
Абсолютное значение	`fabs()`	`cabs()`

Таблица 17.5. Функции для действительных типов с плавающей точкой

Математическая функция	Функция языка C
Арктангенс от частного	`atan2()`
Экспоненциальные функции	`exp2()`, `expm1()`, `frexp()`, `ldexp()`, `scalbn()`, `scalbln()`
Логарифмические функции	`log10()`, `log2()`, `log1p()`, `logb()`, `ilogb()`
Корни	`cbrt()`, `hypot()`
Функции ошибок для нормального распределения	`erf()`, `erfc()`
Гамма-функция	`tgamma()`, `lgamma()`
Остаток	`fmod()`, `remainder()`, `remquo()`
Разделение целой и дробной частей	`modf()`
Следующее целое	`ceil()`, `floor()`
Следующее представимое число	`nextafter()`, `nexttoward()`
Функции округления	`trunc()`, `round()`, `lround()`, `llround()`, `nearbyint()`, `rint()`, `lrint()`, `llrint()`
Положительная разность	`fdim()`
Умножение и сложение	`fma()`
Минимум и максимум	`fmin()`, `fmax()`
Присваивание знака одного числа другому	`copysign()`
Генерация NaN	`nan()`

Таблица 17.6. Функции для комплексных типов с плавающей точкой

Математическая функция	Функция языка C
Выделение действительной и мнимой частей	`creal()`, `cimag()`
Аргумент (угол в полярных координатах)	`carg()`
Сопряжение	`conj()`
Проекция на сферу Римана	`cproj()`

Макросы

Стандартные заголовочные файлы `math.h` и `tgmath.h` определяют ряд макросов, которые могут быть вызваны с аргументами разных типов с плавающей точкой. Различные типы аргументов в одном вызове в языке программирования C поддерживаются только в макросах, но не в вызовах функций.

Макросы, не зависящие от типа

Каждая математическая функция с плавающей точкой существует в трех или шести версиях: по одной для каждого из трех действительных типов или по одной для каждого из трех комплексных типов, или и для действительных, и для комплексных типов. Заголовочный файл `tgmath.h` определяет *макросы, не зависящие от типа*, которые позволяют вызывать необходимую версию данной функции с помощью одного и того же имени. Компилятор обнаруживает подходящую функцию на основе типа переданных аргументов. Таким образом, вам не нужно редактировать вызовы математических функций в своих программах при изменении типа аргумента, например, с `double` на `long double`. Макросы, не зависящие от типа, описаны в разделе "`tgmath.h`" главы 16, "Стандартные заголовочные файлы".

Категории значений с плавающей точкой

C99 определяет пять разновидностей значений действительных типов с плавающей точкой с различными целочисленными макросами для их обозначения (см. раздел "`math.h`" главы 16, "Стандартные заголовочные файлы"):

```
FP_ZERO    FP_NORMAL    FP_SUBNORMAL    FP_INFINITE    FP_NAN
```

Эти классифицирующие макросы, а также некоторые другие макросы, приведенные в табл. 17.7, определены в заголовочном файле `math.h`. Аргументом каждого вызываемого макроса должно быть выражение с действительным типом с плавающей точкой.

Таблица 17.7. Макросы для классификации значений с плавающей точкой

Предназначение	Макросы
Получение категории значения с плавающей точкой	`fpclassify()`
Проверка, принадлежит ли значение с плавающей точкой определенной категории	`isfinite()`, `isinf()`, `isnan()`, `isnormal()`, `signbit()`

Например, два следующих теста эквивалентны:

```
if ( fpclassify( x ) == FP_INFINITE ) /* ... */ ;
if ( isinf( x ) ) /* ... */ ;
```

Макросы сравнения

Можно сравнивать любые два действительных конечных числа с плавающей точкой. Другими словами, одно значение всегда меньше другого, равно ему или больше

него. Однако если один или оба операнда оператора сравнения являются NaN (значением с плавающей точкой, которое не является числом), то такие операнды не являются сравниваемыми. В этом случае операция сравнения всегда дает значение 0, или `false`, и может вызвать исключение с плавающей точкой `FE_INVALID`.

На практике вы можете захотеть избежать риска генерации исключения при сравнении объектов с плавающей точкой. По этой причине заголовочный файл `math.h` определяет макросы, перечисленные в табл. 17.8. Эти макросы дают тот же результат, что и соответствующие выражения с операторами сравнения, но выполняют "тихое" сравнение, т.е. никогда не генерируют исключения, а просто возвращают значение `false`, если операнды не сравнимы. Оба аргумента каждого макроса должны представлять собой выражения с действительными типами с плавающей точкой.

Таблица 17.8. Макросы для сравнения значений с плавающей точкой

Сравнение	Макрос
`(x) > (y)`	`isgreater(x,y)`
`(x) >= (y)`	`isgreaterequal(x,y)`
`(x) < (y)`	`isless(x,y)`
`(x) <= (y)`	`islessequal(x,y)`
`((x) < (y) || (x) > (y))`	`islessgreater(x,y)`*
Проверка на сравнимость	`isunordered(x,y)`

* В отличие от соответствующего операторного выражения макрос `islessgreater()` вычисляет свои аргументы только один раз.

Директивы `#pragma` для арифметических операций

На способ компиляции арифметических выражений влияют две следующие стандартные прагмы:

```
#pragma STDC FP_CONTRACT      on_off_switch
#pragma STDC CX_LIMITED_RANGE on_off_switch
```

Значением *`on_off_switch`* должно быть `ON`, `OFF` или `DEFAULT`. В случае `ON` первая из этих прагм, `FP_CONTRACT`, позволяет компилятору сокращать выражения с плавающей точкой с несколькими операторами C до меньшего количества машинных команд, если это возможно. Такие сокращенные выражения выполняются быстрее. Однако в связи с тем, что они устраняют также ошибки округления, результаты вычислений могут не совпадать в точности с результатами несокращенных выражений. Кроме того, несокращенные выражения могут генерировать исключения с плавающей точкой, которые не будут генерироваться при сокращенных вычислениях. Как именно выполнять сокращения и использовать ли их по умолчанию, зависит от конкретного компилятора.

Вторая прагма, `CX_LIMITED_RANGE`, влияет на умножение, деление и вычисление абсолютных значений комплексных чисел. Эти операции могут вызывать проблемы,

если их операнды бесконечны или если они приводят к переполнению или потере значимости. При использовании значения ON прагма CX_LIMITED_RANGE указывает компилятору, что для этих трех операций можно безопасно использовать простые арифметические методы, так как будут использоваться только конечные операнды, а переполнения и потери значимости можно не обрабатывать. Значение этой прагмы по умолчанию — OFF.

В исходном тексте эти прагмы могут располагаться вне всех функций или в начале блока, до любых объявлений или инструкций. Действие этих прагм начинается с точек, в которых они находятся в исходном тексте. Если прагма располагается вне всех функций, ее действие завершается при обнаружении другой директивы той же самой прагмы или в конце единицы трансляции. Если прагма располагается в блоке, ее действие завершается при обнаружении другой директивы той же самой прагмы во вложенном блоке или в конце содержащего прагму блока. В конце блока компилятор возвращается к состоянию поведения, в котором он находился в начале блока.

Среда вычислений с плавающей точкой

Среда вычислений с плавающей точкой (floating-point environment) включает системные переменные для флагов состояния и режимов управления вычислениями с плавающей точкой. Флаги состояния устанавливаются операциями, которые генерируют *исключения с плавающей точкой* (floating-point exceptions), такие как деление на нуль. Режимы управления представляют собой возможности настройки поведения арифметики с плавающей точкой, которые может устанавливать программа, например каким образом должно выполняться округление результатов вычислений до представимых значений. Поддержка исключений с плавающей точкой и режимов управления является необязательной.

Все объявления, связанные со средой вычислений с плавающей точкой, содержатся в заголовочном файле fenv.h (см. главу 16, “Стандартные заголовочные файлы”).

Программы, осуществляющие доступ к среде вычислений с плавающей точкой, должны заранее информировать о своих намерениях компилятор с помощью следующей стандартной прагмы:

```
#pragma STDC FENV_ACCESS ON
```

Эта директива предупреждает применение компилятором оптимизаций, таких как изменения порядка вычислений, которые могут перемежаться с запросами флагов состояний или применениями режимов управления.

Прагма FENV_ACCESS может применяться так же, как и прагмы FP_CONTRACT и CX_LIMITED_RANGE: вне всех функций или локально в пределах блока (см. предыдущий раздел). Является ли ON или OFF значением прагмы FENV_ACCESS по умолчанию, зависит от компилятора.

Доступ к флагам состояния

Функции, перечисленные в табл. 17.9, обеспечивают доступ к флагам состояний исключений. Единственный аргумент этих функций указывает, какой вид или виды исключений вас интересуют. В заголовочном файле `fenv.h` определены следующие целочисленные макросы, обозначающие типы отдельных исключений:

```
FE_DIVBYZERO    FE_INEXACT    FE_INVALID    FE_OVERFLOW    FE_UNDERFLOW
```

Каждый из этих макросов определен только в том случае, если реализация поддерживает соответствующее исключение. Макрос `FE_ALL_EXCEPT` обозначает все поддерживаемые типы исключений.

Таблица 17.9. Функции для доступа к исключениям с плавающей точкой

Назначение	Функция
Проверка исключения с плавающей точкой	`fetestexcept()`
Сброс исключения с плавающей точкой	`feclearexcept()`
Генерация исключения с плавающей точкой	`feraiseexcept()`
Сохранение исключения с плавающей точкой	`fegetexceptflag()`
Восстановление исключения с плавающей точкой	`fesetexceptflag()`

Режимы округления

Среда вычислений с плавающей точкой включает также режим округления, действующий в настоящее время для операций с плавающей точкой. Заголовочный файл `fenv.h` определяет различные целочисленные макросы для каждого поддерживаемого режима округления. Каждый из приведенных далее макросов определен только тогда, когда реализация поддерживает соответствующее округление:

```
FE_DOWNWARD    FE_TONEAREST    FE_TOWARDZERO    FE_UPWARD
```

Реализации могут определять и другие режимы округления и макросы для их обозначения. Значения этих макросов используются в качестве возвращаемых значений или аргументов функций, перечисленных в табл. 17.10.

Таблица 17.10. Функции режимов округления

Назначение	Функция
Получение текущего режима округления	`fegetround()`
Установка нового режима округления	`fesetround()`

Сохранение среды вычислений с плавающей точкой

Функции, перечисленные в табл. 17.11, работают со средой вычислений с плавающей точкой как с единым целым, позволяя сохранять и восстанавливать состояние этой среды.

Таблица 17.11. Функции для работы со средой вычислений с плавающей точкой

Назначение	Функция
Сохранение среды вычислений с плавающей точкой	`fegetenv()`
Восстановление среды вычислений с плавающей точкой	`fesetenv()`
Сохранение среды вычислений с плавающей точкой и переключение в *неостанавливающий* режим работы*	`feholdexcept()`
Восстановление сохраненной среды и генерации всех установленных в настоящее время исключений	`feupdateenv()`

* При неостанавливающем режиме работы, активизируемом вызовом `feholdexcept()`, исключения с плавающей точкой не прерывают выполнения программы.

Обработка ошибок

Стандарт C99 определяет поведение функций, объявленных в заголовочном файле `math.h`, в случаях некорректных аргументов или математических результатов, выходящих за границы диапазона. Значение макроса `math_errhandling`, которое является константой на протяжении времени выполнения программы, указывает, как программа должна обрабатывать ошибки: с использованием глобальной переменной ошибки `errno`, с применением флагов исключений или обоими способами.

Ошибка области определения

Ошибка области определения происходит, когда математически функция не определена для данного значения аргумента. Например, действительная функция вычисления квадратного корня `sqrt()` не определена для отрицательных значений аргумента. Область определения каждой функции из заголовочного файла `math.h` указана в описании этой функции в главе 18, "Функции стандартной библиотеки".

В случае ошибки области определения функции возвращают значение, определенное реализацией. Кроме того, если выражение `math_errhandling & MATH_ERRNO` не равно нулю (другими словами, если это выражение имеет значение `true`), то функция, вызвавшая ошибку, устанавливает значение переменной ошибки `errno` равным `EDOM`. Если истинно выражение `math_errhandling & MATH_ERREXCEPT`, то функция генерирует исключение с плавающей точкой `FE_INVALID`.

Ошибка диапазона значений

Ошибка диапазона значений возникает тогда, когда математический результат функции не представим возвращаемым типом функции без существенной ошибки округления. *Переполнение* (overflow) возникает из-за того, что математический результат функции имеет конечное значение, являющееся слишком большим для представления возвращаемым типом функции. Если при возникновении переполнения действует режим округления по умолчанию или если точный результат равен бесконечности, то функция возвращает значение `HUGE_VAL` (`HUGE_VALF` или `HUGE_VALL`,

если возвращаемый тип функции — float или long double) с соответствующим знаком. Кроме того, если выражение math_errhandling & MATH_ERRNO истинно, то функция устанавливает значение переменной ошибки errno равным ERANGE. Если истинно выражение math_errhandling & MATH_ERREXCEPT, то переполнение приводит к исключению FE_OVERFLOW, если математический результат функции конечен, и к исключению FE_DIVBYZERO, если он бесконечен.

Потеря значимости (underflow) происходит тогда, когда ошибка выхода за границы диапазона связана с тем, что математически ненулевой результат слишком мал для представления возвращаемым типом функции. При потере значимости функция возвращает значение, которое определяется реализацией, но не превышает значения DBL_MIN (FLT_MIN или LDBL_MIN, в зависимости от типа возвращаемого функцией значения). Реализация также определяет, устанавливает ли в таком случае функция значение переменной ошибки errno равным ERANGE при истинности выражения math_errhandling & MATH_ERRNO. Кроме того, реализация определяет, генерируется ли при потере значимости исключение FE_UNDERFLOW при истинности выражения math_errhandling & MATH_ERREXCEPT.

Классификация и преобразование символов

Стандартная библиотека предоставляет ряд функций для классификации символов и их преобразований. Заголовочный файл ctype.h объявляет такие функции для однобайтных символов с кодами от 0 до 255. В заголовочном файле wctype.h объявлены аналогичные функции для широких символов, тип которых — wchar_t. Эти функции обычно реализованы в виде макросов.

Результаты этих функций, за исключением isdigit() и isxdigit(), зависят от текущих установок локали для категории LC_CTYPE. Вы можете запросить или установить локаль с помощью функции setlocale().

Классификация символов

Функции, перечисленные в табл. 17.12, выполняют проверку принадлежности символов определенной категории. Возвращаемое значение функций — ненулевое (true), когда аргумент представляет собой код символа, относящегося к данной категории.

Таблица 17.12. Функции классификации символов

Категория	Функции в ctype.h	Функции в wctype.h
Буквы	isalpha()	iswalpha()
Строчные буквы	islower()	iswlower()
Прописные буквы	isupper()	iswupper()
Десятичные цифры	isdigit()	iswdigit()

Окончание табл. 17.12

Категория	Функции в `ctype.h`	Функции в `wctype.h`
Шестнадцатеричные цифры	`isxdigit()`	`iswxdigit()`
Буквы и десятичные цифры	`isalnum()`	`iswalnum()`
Печатаемые символы (включая пробельные)	`isprint()`	`iswprint()`
Печатаемые непробельные символы	`isgraph()`	`iswgraph()`
Пробельные символы	`isspace()`	`iswspace()`
Пробельные символы, разделяющие слова в строке текста	`isblank()`	`iswblank()`
Символы препинания	`ispunct()`	`iswpunct()`
Управляющие символы	`iscntrl()`	`iswcntrl()`

Функции `isgraph()` и `iswgraph()` ведут себя по-разному, если выполнимый набор символов содержит дополнительные кодируемые байтами печатаемые пробельные символы (т.е. пробельные символы, не являющиеся управляющими) в дополнение к символу пробела (`' '`). В этом случае `iswgraph()` для таких печатаемых пробельных символов возвращает значение `false`, в то время как `isgraph()` возвращает `false` только для символа пробела (`' '`).

Заголовочный файл `wctype.h` объявляет также две дополнительные функции, перечисленные в табл. 17.13, для тестирования широких функций. Они называются *расширяемыми* (extensible) функциями классификации, которые вы можете использовать для проверки, принадлежит ли широкий символ категории, определенной реализацией и задаваемой строкой.

Таблица 17.13. Расширяемые функции классификации символов

Назначение	Функция
Отображает строковый аргумент, указывающий класс символов, на скалярное значение, которое может использоваться в качестве второго аргумента `iswctype()`	`wctype()`
Проверяет, принадлежит ли широкий символ классу, указываемому вторым аргументом	`iswctype()`

Эти две функции из табл. 17.13 могут применяться для выполнения как минимум тех же тестов, что и функции из табл. 17.12. Строки, которые обозначают классы символов, распознаваемые функцией `wctype()`, формируются из имен соответствующих тестовых функций с удаленным префиксом `isw`. Например, строка `"alpha"`, подобно имени функции `iswalpha()`, обозначает категорию “буквы”. Таким образом, для значения *wc*, являющегося широким символом, следующие проверки эквивалентны:

```
iswalpha(wc)
iswctype(wc, wctype("alpha"))
```

Реализации могут определять и другие такие строки для обозначения специфичных для конкретной локали классов символов.

Изменение регистра

Функции, перечисленные в табл. 17.14, дают символ в верхнем регистре, соответствующий данному символу в нижнем регистре, и наоборот. Все прочие аргументы возвращаются неизмененными.

Таблица 17.14. Функции преобразования символов

Преобразование регистра	Функции в `ctype.h`	Функции в `wctype.h`
Верхний в нижний	`tolower()`	`towlower()`
Нижний в верхний	`toupper()`	`towupper()`

Как и в предыдущем разделе, заголовочный файл `wctype.h` объявляет две дополнительные *расширяемые функции* для преобразования широких символов. Эти функции приведены в табл. 17.15. Каждое поддерживаемое данной реализацией преобразование задается с помощью строки.

Таблица 17.15. Расширяемые функции преобразования символов

Назначение	Функция
Отображает строковый аргумент, описывающий преобразование символов, на скалярное значение, которое может быть использовано как второй аргумент функции `towctrans()`	`wctrans()`
Выполняет преобразование данного широкого символа, указанное вторым аргументом	`towctrans()`

Две функции из табл. 17.15 могут использоваться как минимум для выполнения тех же преобразований, которые осуществляются функциями, перечисленными в табл. 17.14. Строки, обозначающие эти преобразования, представляют собой `"tolower"` и `"toupper"`. Таким образом, для широкого символа *wc* следующие вызовы дают один и тот же результат:

```
towupper(wc);
towctrans(wc, wctrans("toupper"));
```

Реализации могут определять и другие строки для преобразований символов, специфичных для данной локали.

Работа со строками

Строка представляет собой непрерывную последовательность символов, завершающуюся нулевым символом `'\0'` — символом завершения строки. Длина строки равна количеству символов до завершающего нулевого символа. Строки могут быть *байтными*, состоящими из символов размером один байт каждый, или *широкими*, состоящими из широких символов. Байтные строки хранятся в массивах с элементами типа `char`, а широкие строки — в массивах, элементы которых имеют один из широкосимвольных типов: `wchar_t`, `char16_t` или `char32_t`.

В языке C нет фундаментального типа для строк, а следовательно, нет и операторов для конкатенации, сравнения или присваивания значений строкам. Вместо этого стандартная библиотека предоставляет ряд функций, перечисленных в табл. 17.16, которые выполняют указанные и иные операции со строками. В заголовочном файле `string.h` объявлены функции для обычных строк, состоящих из символов типа `char`. Имена этих функций начинаются со `str`. В заголовочном файле `wchar.h` объявлены соответствующие функции для строк, состоящих из широких символов; их имена начинаются с `wcs`.

Подобно прочим массивам, строки, встречающиеся в выражениях, неявно преобразуются в указатель на их первый элемент. Таким образом, когда вы передаете в функцию в качестве аргумента строку, функция получает только указатель на первый символ и может определить длину строки только по положению завершающего нулевого символа.

Таблица 17.16. Функции для работы со строками

Назначение	Функции в `string.h`	Функции в `wchar.h`
Поиск длины строки	`strlen()`, `strnlen_s()`	`wcslen()`, `wcsnlen_s()`
Копирование строк	`strcpy()`, `strncpy()`, `strcpy_s()`, `strncpy_s()`	`wcscpy()`, `wcsncpy()`, `wcscpy_s()`, `wcsncpy_s()`
Конкатенация строк	`strcat()`, `strncat()`, `strcat_s()`, `strncat_s()`	`wcscat()`, `wcsncat()`, `wcscat_s()`, `wcsncat_s()`
Сравнение строк	`strcmp()`, `strncmp()`, `strcoll()`	`wcscmp()`, `wcsncmp()`, `wcscoll()`
Преобразование строки, такое, что сравнение двух преобразованных строк с помощью `strcmp()` дает тот же результат, что и сравнение исходных строк с помощью функции `strcoll()`, учитывающей текущую локаль	`strxfrm()`	`wcsxfrm()`
Поиск в строке:		
…первое или последнее появление данного символа	`strchr()`, `strrchr()`	`wcschr()`, `wcsrchr()`
…первое вхождение другой строки	`strstr()`	`wcsstr()`
…первое появление любого символа из данного множества	`strcspn()`, `strpbrk()`	`wcscspn()`, `wcspbrk()`
…первое появление любого символа, не являющегося членом данного множества	`strspn()`	`wcsspn()`
Разбиение строки на токены	`strtok()`, `strtok_s()`	`wcstok()`, `wcstok_s()`

Многобайтные символы

В *многобайтных наборах символов* каждый символ кодируется как последовательность из одного или нескольких байтов (см. раздел "Широкие символы и многобайтные символы" в главе 1, "Основы языка C"). В то время как каждый широкий символ представлен одним объектом типа `wchar_t`, `char16_t` или `char32_t`, количество байтов, необходимых для представления данного символа в многобайтной кодировке, является величиной переменной. Однако количество байтов в представлении многобайтного символа, включая любые необходимые служебные последовательности, никогда не превышает значение макроса `MB_CUR_MAX`, определенного в заголовочном файле `stdlib.h`.

Функции стандартной библиотеки позволяют получать код широкого символа, соответствующего любому многобайтному символу, и многобайтное представление любого широкого символа. Некоторые схемы многобайтного кодирования являются *схемами с состояниями* (зависящими от предыстории); интерпретация данной многобайтной последовательности может зависеть от ее положения по отношению к управляющим символам, именуемым *последовательностями сдвигов* (shift sequences), которые используются в многобайтном потоке или строке. В таких ситуациях преобразование многобайтного символа в широкий символ или многобайтной строки в широкую строку зависит от текущего *состояния сдвига* в точке считывания первого многобайтного символа. По той же причине преобразование широкого символа в многобайтный или широкой строки в многобайтную может повлечь за собой вставку соответствующих последовательностей сдвигов в выходную последовательность. Примером многобайтной схемы кодирования, которая применяет последовательность сдвигов, является BOCU-1, совместимая с MIME сжатая схема кодирования Unicode, требующая меньшего пространства, чем UTF-8. Схема же UTF-8 последовательности сдвигов не использует.

Преобразование между широкими и многобайтными символами или строками может быть необходимым, когда вы читаете или записываете символы в поток, *ориентированный на широкие символы* (см. раздел "Байтные и широкие потоки" главы 13, "Ввод-вывод").

В табл. 17.17 перечислены все функции стандартной библиотеки для работы с многобайтными символами.

Таблица 17.17. Функции для работы с многобайтными символами

Назначение	Функции в `stdlib.h`	Функции в `wchar.h`	Функции в `uchar.h`
Длина многобайтного символа	`mblen()`	`mbrlen()`	
Широкий символ, соответствующий данному многобайтному символу	`mbtowc()`	`mbrtowc()`	`mbrtoc16()`, `mbrtoc32()`
Многобайтный символ, соответствующий данному широкому символу	`wctomb()`, `wctomb_s()`	`wcrtomb()`, `wcrtomb_s()`	`c16rtomb()`, `c32rtomb()`

Окончание табл. 17.17

Назначение	Функции в stdlib.h	Функции в wchar.h	Функции в uchar.h
Преобразование многобайтной строки в широкую	mbstowcs(), mbstowcs_s()	mbsrtowcs(), mbsrtowcs_s()	
Преобразование широкой строки в многобайтную	wcstombs(), wcstombs_s()	wcsrtombs(), wcsrtombs_s()	
Преобразования между однобайтными символами и широкими символами		btowc(), wctob()	
Проверка начального состояния сдвига		mbsinit()	

Буква r в именах функций, объявленных в заголовочном файле wchar.h, означает "повторно запускаемая" (restartable). Такие функции, в противоположность функциям, объявленным без r в имени в заголовочном файле stdlib.h, получают дополнительный аргумент, который представляет собой указатель на объект, хранящий состояние сдвига многобайтного символа или строкового аргумента.

Преобразования между числами и строками

Стандартная библиотека предоставляет ряд функций для анализа числовой строки и возврата числового значения. Эти функции перечислены в табл. 17.18. Функции преобразования отличаются как целевыми типами, так и типами строк, которые они преобразуют. Функции для строк символов типа char объявлены в заголовочном файле stdlib.h, а для широких строк — в заголовочном файле wchar.h. Кроме того, C99 вводит четыре функции для преобразования строк в знаковый и беззнаковый целочисленный тип наибольшего имеющегося размера, intmax_t или uintmax_t. Эти четыре функции объявлены в заголовочном файле inttypes.h.

Таблица 17.18. Преобразование числовых строк

Преобразование	Функции в stdlib.h	Функции в wchar.h	Функции в inttypes.h
Строка в int	atoi()		
Строка в long	atol(), strtol()	wcstol()	
Строка в unsigned long	strtoul()	wcstoul()	
Строка в long long	atoll(), strtoll()	wcstoll()	
Строка в unsigned long long	strtoull()	wcstoull()	
Строка в intmax_t			strtoimax(), wcstoimax()
Строка в uintmax_t			strtoumax(), wcstoumax()
Строка в float	strtof()	wcstof()	
Строка в double	atof(), strtod()	wcstod()	
Строка в long double	strtold()	wcstold()	

Функции strtol(), strtoll() и strtod() могут быть более полезными при использовании, чем функции atol(), atoll() и atof(), так как они возвращают позицию следующего символа в исходной строке после того, как последовательность символов интерпретирована как число.

В дополнение к функциям, перечисленным в табл. 17.18, можно выполнять преобразования строк в числа с помощью одной из функций семейства sscanf() с соответствующей строкой формата. Аналогично можно использовать функции sprintf() для выполнения обратных преобразований, генерируя числовую строку из числового аргумента. Эти функции объявлены в заголовочном файле stdio.h. Опять же, соответствующие функции для широких строк объявлены в заголовочном файле wchar.h. Оба набора функций перечислены в табл. 17.19.

Таблица 17.19. Преобразования между числами и строками с использованием строк формата

Преобразование	Функции в stdio.h	Функции в wchar.h
Строка в число	sscanf(), vsscanf()	swscanf(), vswscanf()
Число в строку	sprintf(), snprintf(), vsprintf(), vsnprintf()	swprintf(), vswprintf()

Для каждой из этих функций имеется безопасная альтернативная функция, имя которой заканчивается суффиксом _s.

Поиск и сортировка

В табл. 17.20 перечислены четыре функции стандартной библиотеки, объявленные в заголовочном файле stdlib.h. Функции для поиска в строках перечислены в разделе “Работа со строками” данной главы.

Таблица 17.20. Функции для поиска и сортировки

Назначение	Функция
Сортировка массива	qsort(), qsort_s()
Поиск в отсортированном массиве	bsearch(), bsearch_s()

Эти функции предоставляют абстрактный интерфейс, который позволяет использовать их для массивов с любым типом элементов. Одним из параметров функций qsort() и qsort_s() является указатель на функцию обратного вызова, которую функции qsort() и qsort_s() могут использовать для сравнения пар элементов массива. Обычно эти функции необходимо определять самостоятельно. Функции bsearch() и bsearch_s(), которые выполняют поиск элемента массива, указанного с помощью аргумента-“ключа”, применяют ту же самую методику вызова пользовательской функции для сравнения элементов массива с заданным ключом.

Функции bsearch() и bsearch_s() используют алгоритм бинарного поиска, так что массив, в котором выполняется поиск, должен быть отсортирован. Хотя имена функций qsort() и qsort_s() и предполагают применение алгоритма быстрой сортировки, стандарт не определяет, какой именно алгоритм должен быть использован.

Работа с блоками памяти

Функции, перечисленные в табл. 17.21, инициализируют, копируют, сравнивают блоки памяти и выполняют в них поиск. Эти функции, объявленные в заголовочном файле string.h, обращаются к блокам памяти байт за байтом, а объявленные в заголовочном файле wchar.h читают и записывают элементы типа wchar_t. Соответственно, параметр размера каждой функции указывает размер блока памяти либо как количество байтов, либо как количество широких символов.

Таблица 17.21. Функции для работы с блоками памяти

Назначение	Функции в string.h	Функции в wchar.h
Копирует блок памяти, когда исходный и целевой блоки не перекрываются	memcpy(), memcpy_s()	wmemcpy(), wmemcpy_s()
Копирует блок памяти, когда исходный и целевой блоки могут перекрываться	memmove(), memmove_s()	wmemmove(), wmemmove_s()
Сравнивает два блока памяти	memcmp()	wmemcmp()
Находит первое вхождение данного символа	memchr()	wmemchr()
Заполняет блок памяти указанным значением	memset(), memset_s()	wmemset(), wmemset_s()

Управление динамической памятью

Многие программы, включая, например, работающие с динамическими структурами данных, зависят от возможности выделения и освобождения блоков памяти во время выполнения. Программы на языке программирования C могут выполнять эти действия с помощью четырех функций для работы с динамической памятью, объявленных в заголовочном файле stdlib.h и перечисленных в табл. 17.22. Использование этих функций детально описано в главе 12, “Управление динамической памятью”.

Таблица 17.22. Функции для работы с динамической памятью

Назначение	Функция
Выделение блока памяти	malloc()
Выделение блока памяти и обнуление его содержимого	calloc()
Изменение размера блока памяти	realloc()
Освобождение блока памяти	free()

Дата и время

В заголовочном файле `time.h` объявлены функции стандартной библиотеки, предназначенные для получения текущих даты и времени, получения времени работы процесса, выполнения ряда преобразований информации о дате и времени и для форматирования этой информации для вывода. Ключевой функцией является `time()`, которая возвращает текущее календарное время в виде арифметического значения типа `time_t`. Обычно оно представляет собой количество секунд, прошедшее с определенного момента в прошлом, именуемого *эпохой*. Эпоха Unix представляет собой момент времени 1 января 1970 года, соответствующий времени 00:00:00 UTC (Универсальное глобальное время (Coordinated Universal Time), ранее именовавшееся Гринвичским средним временем (Greenwich Mean Time — GMT)). Имеются также стандартные функции для преобразования значения календарного времени с типом `time_t` в строку или структуру типа `struct tm`. Этот структурный тип имеет члены типа `int` для секунд, минут, часов, дня, месяца, года, дня недели, дня года и флага летнего времени (см. описание функции `gmtime()` в главе 18, "Функции стандартной библиотеки"). Функции для работы с датой и временем перечислены табл. 17.23.

Таблица 17.23. Функции для работы с датой и временем

Назначение	Функция
Получение использованного процессорного времени	`clock()`
Получение текущего календарного времени	`time()`
Получение разницы между двумя календарными временами	`difftime()`
Преобразование календарного времени в `struct tm`	`gmtime()`, `gmtime_s()`
Преобразование календарного времени в `struct tm` с локальными значениями времени	`localtime()`, `localtime_s()`
Нормализация значений объекта `struct tm` и возврат календарного времени с типом `time_t`	`mktime()`
Преобразование календарного времени в строку	`ctime()`, `ctime_s()`, `asctime()`, `asctime_s()`, `strftime()`, `wcsftime()`

Исключительно гибкая функция `strftime()` использует для генерации строки с информацией о дате и времени строку формата (подобно функциям семейства `printf()`) и категорию локали `LC_TIME`. Запросить или изменить действующую локаль можно с помощью функции `setlocale()`. Функция `wcsftime()` представляет собой версию функции `strftime()` для широких строк; эта функция объявлена в заголовочном файле `wchar.h`.

Диаграмма на рис. 17.1 подытоживает доступные в языке программирования функции для работы с датой и временем.

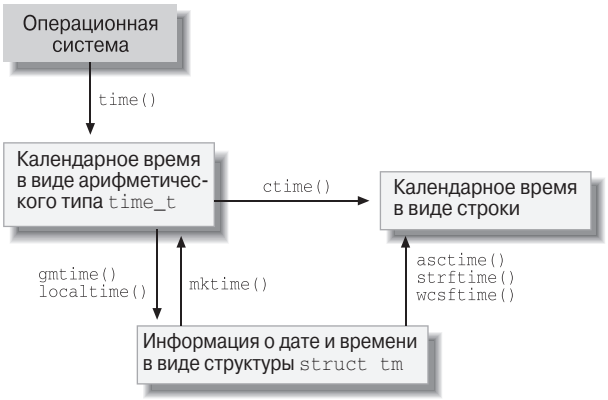

Рис. 17.1. Функции для работы с датой и временем

Управление процессом

Процесс представляет собой выполняемую программу. Каждый процесс имеет ряд атрибутов, таких, например, как открытые файлы. Конкретные атрибуты процесса зависят от данной операционной системы. Возможности стандартной библиотеки по управлению процессами можно разделить на два вида: предназначенные для обмена информацией с операционной системой и работающие с сигналами.

Обмен информацией с операционной системой

Функции из табл. 17.24 объявлены в заголовочном файле `stdlib.h` и позволяют программе обмениваться информацией с операционной системой.

Таблица 17.24. Функции для обмена информацией с операционной системой

Назначение	Функция
Запрос значения переменной среды	`getenv()`, `getenv_s()`
Выполнение системной команды	`system()`
Регистрация функции, выполняемой при завершении программы	`atexit()`, `at_quick_exit()`
Нормальное завершение программы	`exit()`, `_Exit()`, `quick_exit()`
Аварийное завершение программы	`abort()`

В Unix и Windows одним из атрибутов процесса является *среда*, которая состоит из списка строк вида *имя=значение*. Обычно процесс наследует среду, сгенерированную родительским процессом. Функция `getenv()` представляет собой единственный способ получения программой управляющей информации, такой как имена каталогов, содержащих используемые файлы.

В противоположность функции `exit()` функция `_Exit()` игнорирует все сигналы и не вызывает никакие зарегистрированные с помощью вызова `atexit()` функции. Функция `quick_exit()`, введенная стандартом C99, сначала вызывает все функции, зарегистрированные функцией `at_quick_exit()`, а затем завершает программу с помощью вызова `_Exit()`. Функция `abort()` вызывает аварийное завершение программы путем генерации сигнала `SIGABRT`.

Сигналы

Операционная система посылает процессам различные сигналы, уведомляя их о необычных событиях. К подобным событиям обычно относятся серьезные ошибки, например некорректное обращение к памяти или аппаратные прерывания, такие как сигналы от таймера. Кроме того, сигналы могут быть сгенерированы пользователем с консоли и самой программой — с помощью вызова функции `raise()`.

Каждая программа может определить, как она должна реагировать на конкретные сигналы. Программа может игнорировать сигнал, позволить обработчику сигнала по умолчанию самому разобраться с ним или установить собственную функцию в качестве *обработчика сигнала*. Обработчик сигнала представляет собой функцию, которая выполняется автоматически, когда программа получает сигнал данного типа.

Две функции C, работающие с сигналами, а также макросы, обозначающие типы сигналов, объявлены в заголовочном файле `signal.h`. Эти функции перечислены в табл. 17.25.

Таблица 17.25. Функции для работы с сигналами

Назначение	Функция
Установка обработки сигнала данного типа	`signal()`
Отправление сигнала вызывающему процессу	`raise()`

Интернационализация

Стандартная библиотека поддерживает разработку программ на языке программирования C, которые могут адаптироваться к местным культурным соглашениям. Например, программы могут использовать специфичные для данного местонахождения наборы языковых символов или форматы для денежной информации.

Все программы начинают работу в локали по умолчанию с именем `"C"`, которая не содержит информацию о стране или языке. Во время выполнения программы можно изменить текущую локаль или запросить информацию о текущей локали. Информация, составляющая локаль, подразделена на категории, которые можно запрашивать и устанавливать по отдельности.

Функции, которые работают с текущей локалью, а также соответствующие типы и макросы объявлены в заголовочном файле `locale.h`. Они перечислены в табл. 17.26.

Таблица 17.26. Функции для работы с локалями

Назначение	Функция
Запрос или установка локали для определенной категории	`setlocale()`
Получение информации о соглашениях форматирования для числовых и денежных строк	`localeconv()`

Многие функции используют информацию, специфичную для данной локали. Описания стандартных библиотечных функций в главе 18, "Функции стандартной библиотеки", указывают, какие функции обращаются к информации о локали. Эти функции включают следующие:

- функции классификации и преобразования регистров символов;
- сравнение строк с учетом локали (`strcoll()` и `wcscoll()`);
- форматирование даты и времени (`strftime()` и `wcsftime()`);
- преобразование числовых строк;
- преобразования между многобайтными и широкими символами.

Нелокальные переходы

Инструкция `goto` в языке программирования C может использоваться только для организации безусловных переходов в пределах функций. Бо́льшую свободу обеспечивает пара функций, объявленных в заголовочном файле `setjmp.h`, обеспечивающая возможность перехода в любую точку программы. Эти функции перечислены в табл. 17.27.

Таблица 17.27. Функции нелокальных переходов

Назначение	Функция
Сохраняет текущий контекст выполнения в качестве целевого местоположения для функции `longjmp()`	`setjmp()`
Переход в программный контекст, сохраненный вызовом функции `setjmp()`	`longjmp()`

Макрос `setjmp()` при вызове сохраняет в своем аргументе значение типа `jmp_buf`, которое действует как закладка для данной точки программы. Объект `jmp_buf` содержит все необходимые части текущего состояния выполнения (в том числе значения регистров и стека). При передаче объекта типа `jmp_buf` функции `longjmp()` последняя восстанавливает сохраненное состояние, и выполнение

программы продолжается с точки, следующей за соответствующей точкой вызова `setjmp()`. Вызов функции `longjmp()` не должен осуществляться до вызова функции `setjmp()`. Кроме того, если любая переменная с автоматической длительностью хранения в функции, которая вызывает функцию `setjmp()`, была изменена после вызова `setjmp()` (и не была объявлена как `volatile`), то ее значение после вызова `longjmp()` является неопределенным.

Возвращаемое значение `setjmp()` указывает, когда программа достигла этой точки: после исходного вызова `setjmp()` или после вызова `longjmp()`; сам вызов `setjmp()` возвращает 0. Если же `setjmp()` возвращает любое другое значение, значит, эта точка программы была достигнута путем вызова `longjmp()`. Если второй аргумент в `longjmp()` — запрашиваемое возвращаемое значение — равен 0, то он заменяется единицей в качестве кажущегося возвращаемого значения после соответствующего вызова `setjmp()`.

Многопоточность (C11)

Возможности, предоставляемые стандартом C11 для программирования многопоточных приложений, подробно описаны в главе 14, "Многопоточность". В таблицах этого раздела подытожены функции многопоточной библиотеки C. Обратите внимание, что поддержка многопоточности и атомарных операций не является обязательной. Реализация, которая соответствует C11, просто должна определить макросы `__STDC_NO_THREADS__` и `__STDC_NO_ATOMICS__`, если она не обеспечивает соответствующие функциональные возможности.

Функции для работы с потоками

Библиотека для работы с потоками предоставляет функции для решения следующих задач:

- управление потоками;
- синхронизация выполнения потоков с использованием мьютексов;
- использование условных переменных для обмена информацией между потоками;
- локальная по отношению к потоку память.

Соответственно, имена функций для работы с многопоточностью имеют один из префиксов `thrd_`, `mtx_`, `cnd_` и `tss_`. Единственным исключением является функция `call_once()` (см. табл. 17.28). Все эти функции объявлены в заголовочном файле `threads.h`.

Таблица 17.28. Функция инициализации

Назначение	Функция
Вызов функции ровно один раз	`call_once()`

Функция `call_once()` гарантирует, что будет выполнен только первый вызов функции, указанной в качестве аргумента. Это может пригодиться, например, при инициализации данных, совместно используемых несколькими потоками.

Функции, перечисленные в табл. 17.29, выполняют операции над потоками выполнения программы, такие как их запуск или остановка.

Таблица 17.29. Функции для управления потоками

Назначение	Функция
Создание и запуск нового потока для выполнения определенной функции	`thrd_create()`
Получение идентификатора потока, выполняющего вызов этой функции	`thrd_current()`
Проверка, ссылаются ли два идентификатора потока на один и тот же поток	`thrd_equal()`
Приостановка выполнения текущего потока на определенное время	`thrd_sleep()`
Совет системе переключиться на выполнение другого потока	`thrd_yield()`
Завершение текущего потока	`thrd_exit()`
Ожидание завершения другого потока	`thrd_join()`
Отказ от потока	`thrd_detach()`

Язык программирования C предоставляет перечисленные в табл. 17.30 функции для синхронизации работы разных потоков с использованием мьютексов.

Таблица 17.30. Функции для работы с мьютексами

Назначение	Функция
Создание и инициализация мьютекса	`mtx_init()`
Блокировка мьютекса; если занят, ожидание до тех пор, пока не станет свободным	`mtx_lock()`
Блокировка мьютекса, только если он становится доступным в течение определенного времени	`mtx_timedlock()`
Блокировка мьютекса, только если он доступен прямо сейчас	`mtx_trylock()`
Уничтожение указанного мьютекса	`mtx_destroy()`

Переменные условий используются для обмена информацией между различными потоками программы, как, например, когда один поток должен уведомить другие потоки о доступности некоторых данных. В табл. 17.31 перечислены все функции, предназначенные для работы с такими переменными.

Таблица 17.31. Функции для работы с переменными условий

Назначение	Функция
Инициализация переменной условия	`cnd_init()`
Активирует один из потоков, ожидающих переменную условия	`cnd_signal()`
Активирует все потоки, ожидающие переменную условия	`cnd_broadcast()`
Ожидает переменную условия до тех пор, пока не будет активирован иным потоком	`cnd_wait()`
Ожидает переменную условия ограниченное время	`cnd_timedwait()`
Уничтожает переменную условия	`cnd_destroy()`

Четыре функции, перечисленные в табл. 17.32, работают с памятью, специфичной для данного потока (thread-specific storage — TSS). Несколько потоков используют глобальный ключ, который представляет собой указатель каждого потока на блок памяти, специфичный для данного потока.

Таблица 17.32. Функции для работы с памятью, специфичной для данного потока

Назначение	Функция
Создает ключ TSS и (необязательно) определяет деструктор, вызываемый при завершении потока	`tss_create()`
Настраивает блок памяти для текущего потока для доступа с помощью данного ключа	`tss_set()`
Возвращает указатель на блок памяти текущего потока для данного ключа	`tss_get()`
Освобождает ресурсы, использованные ключом TSS	`tss_delete()`

Атомарные операции

Объявления функций и определений типов и макросов для атомарных операций находятся в заголовочном файле `stdatomic.h`.

Макрос `ATOMIC_VAR_INIT` может использоваться для инициализации атомарных объектов. Макрос вида `_ATOMIC_`*`тип`*`_LOCK_FREE` указывает, не обладает ли атомарный объект соответствующего целочисленного *типа* свойством "свободы от блокировок" (детальную информацию можно найти в разделе "`stdatomic.h`" главы 16, "Стандартные заголовочные файлы"). Перечисленные в табл. 17.33 функции могут использоваться в качестве альтернативы этим макросам.

Таблица 17.33. Функции для инициализации атомарных объектов и определения, являются ли атомарные объекты свободными от блокировок

Назначение	Функция
Инициализация атомарного объекта	`atomic_init()`
Проверка, является ли атомарный объект свободным от блокировок	`atomic_is_lock_free()`

Чтение и запись атомарных объектов является атомарной операцией. Операции "чтения-изменения-записи", подобные выполняемым операторами инкремента и декремента (++ и --), а также составными операторами (+= и т.п.) в случае, когда левый операнд является атомарным объектом, также являются атомарными операциями. Однако инициализация атомарного объекта атомарной операцией не является.

Помимо упомянутых операторов, стандартная библиотека предоставляет ряд функций для выполнения атомарных операций, такие как `atomic_load()`. По умолчанию атомарные операции выполняются с наиболее строгим ограничением упорядочения памяти: ограничением *последовательной согласованности*. Для выполнения атомарных операций с более слабым ограничением на упорядочение памяти имеется вторая версия каждой функции атомарной операции, которая принимает дополнительный аргумент, явно указывающий ограничение упорядочения памяти. Эти функции имеют имена, которые заканчиваются на `_explicit`, например `atomic_load_explicit()`. Дополнительные сведения об этих функциях можно найти в разделе "Упорядочение памяти" главы 14, "Многопоточность".

Обобщенные функции, перечисленные в табл. 17.34, могут использоваться с объектами всех атомарных типов, которые определены в заголовочном файле `stdatomic.h`. Полный список этих типов имеется в разделе "`stdatomic.h`" главы 16, "Стандартные заголовочные файлы".

Таблица 17.34. Обобщенные функции для атомарных операций

Назначение	Функция
Получение значения атомарного объекта	`atomic_load(),atomic_load_explicit()`
Запись значения атомарного объекта	`atomic_store(),atomic_store_explicit()`
Получение существующего значения и запись нового значения	`atomic_exchange(),atomic_exchange_explicit()`
Сравнение значения атомарного объекта с ожидаемым значением; если они равны, запись нового значения объекта	`atomic_compare_exchange_strong(), atomic_compare_exchange_strong_explicit(), atomic_compare_exchange_weak(), atomic_compare_exchange_weak_explicit()`
Замена значения целочисленного атомарного объекта результатом арифметической или битовой операции; в отличие от соответствующих составных операторов присваивания эти функции возвращают исходное значение объекта до выполнения операции	`atomic_fetch_add(), atomic_fetch_add_explicit(),atomic_fetch_sub(), atomic_fetch_sub_explicit(),atomic_fetch_or(), atomic_fetch_or_explicit(),atomic_fetch_xor(), atomic_fetch_xor_explicit(),atomic_fetch_and(), atomic_fetch_and_explicit()`

Объекты типа atomic_flag гарантированно свободны от блокировок. Функции, перечисленные в табл. 17.35, обеспечивают обычные операции с флагами для объектов atomic_flag.

Таблица 17.35. Функции для объектов типа atomic_flag

Назначение	Функция
Сброс атомарного флага	atomic_flag_clear(), atomic_flag_clear_explicit()
Установка атомарного флага и возврат его предыдущего значения	atomic_flag_test_and_set(), atomic_flag_test_and_set_explicit()

Барьеры памяти определяют ограничения упорядочения памяти, которые должны быть наблюдаемы при синхронизации операций атомарных записи и чтения (см. раздел "Барьеры памяти" главы 14, "Многопоточность").

Таблица 17.36. Функции для работы с барьерами памяти

Назначение	Функция
Вставка барьера выборки, освобождения или выборки и освобождения	atomic_thread_fence()
Вставка барьера, который применяет ограничения упорядочения только между операциями в потоке и в обработчике сигнала, выполняемом в этом потоке	atomic_signal_fence()

Отладка

Использование макроса assert() — это простой способ поиска логических ошибок в процессе разработки программы. Этот макрос определен в заголовочном файле assert.h. Он просто проверяет, имеет ли его скалярный аргумент ненулевое значение. Если значение этого аргумента равно нулю, макрос assert() выводит сообщение об ошибке, которое содержит выражение аргумента, имя функции, имя файла и номер строки, а затем вызывает функцию abort(), чтобы прекратить работу программы. В следующем примере вызов assert() выполняет некоторую проверку корректности аргумента, передаваемого функции free():

```
#include <stdlib.h>
#include <assert.h>

char *buffers[64] = { NULL };   // Массив указателей

int i;

/* ... выделение буферов памяти и работа с ними ... */

assert( i >= 0 && i < 64 );     // Индекс в допустимых границах?
```

```
assert( buffers[i] != NULL );   // Использовался ли этот указатель?
free( buffers[i] );
```

Вместо того чтобы пытаться освободить несуществующий буфер, этот код прекращает выполнение программы (в данном случае скомпилированной как файл `assert.c`) со следующим диагностическим сообщением:

```
assert:assert.c:14:main: Assertion 'buffers[i] != ((void *)0)' failed.
Aborted
```

По завершении тестирования вы можете отключить все вызовы `assert()` с помощью определения макроса `NDEBUG` до директивы `#include` для заголовочного файла `assert.h`. Этот макрос может не иметь замещающего значения, например

```
#define NDEBUG
#include <assert.h>
/* ... */
```

Стандарт C11 ввел возможность тестирования целочисленных константных выражений во время компиляции. Это делается с помощью объявления `_Static_assert`. Детальное описание этой возможности можно найти в разделе “Объявления _Static_assert” главы 11, “Объявления”.

Сообщения об ошибках

Различные функции стандартной библиотеки устанавливают значение глобальной переменной `errno`, указывающее тип ошибки, возникшей во время выполнения функции (см. раздел “`errno.h`” главы 16, “Стандартные заголовочные файлы”). Функции из табл. 17.37 генерируют для текущего значения `errno` соответствующее сообщение об ошибке.

Таблица 17.37. Функции для сообщения об ошибках

Назначение	Функция	Заголовочный файл
Выводит соответствующее текущему значению `errno` сообщение об ошибке в поток `stderr`	`perror()`	`stdio.h`
Возвращает указатель на сообщение об ошибке, соответствующее данному номеру ошибки	`strerror()`	`string.h`
Копирует соответствующее текущему значению `errno` сообщение об ошибке в массив	`strerror_s()`	`string.h`
Возвращает длину сообщения об ошибке, соответствующего текущему значению `errno`	`strerrorlen_s()`	`string.h`

Функция `perror()` выводит строку, переданную ей в качестве аргумента, двоеточие и сообщение об ошибке, соответствующее значению `errno`. Это сообщение об ошибке то же, что и то, которое функция `strerror()` возвращает при вызове с тем же значением в качестве аргумента, например

```
if (remove("test1") != 0)   // Если мы не можем удалить файл...
    perror( "Не удаляется 'test1'" );
```

Этот вызов `perror()` дает тот же вывод, что и следующая инструкция:

```
fprintf( stderr, "Не удаляется 'test1': %s\n", strerror( errno ) );
```

В этом примере, если файл `test1` не существует, программа, скомпилированная с помощью GCC, выведет следующее сообщение (в переводе):

```
Не удаляется 'test1': нет такого файла или каталога
```

Сообщение об ошибке, адрес которого предоставляет функция `strerror()`, может быть изменено последующими вызовами `strerror()`. Чтобы избежать гонки данных, многопоточные программы должны использовать альтернативную функцию `strerror_s()`, которая копирует сообщение об ошибке в массив, предоставляемый вызывающей функцией.