КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 10 — Библиотека

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

2021.10.13

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 ним 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.

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

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, при этом будет генерироваться 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.

Если для обнаружения ошибок условий используется **assert()**, не рекомендуется определять **NDEBUG**, 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.

Макросы

`NULL`

расширяется до целочисленного константного выражения, обозначающего значение нулевого указателя.

`offsetof(type, `*`обозначение-члена`*`)`

расширяется до целочисленного константного выражения, имеющего тип **`size_t`**, значением которого является смещение в байтах, до элемента структуры *обозначение-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

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

MB_CUR_MAX

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**

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.

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 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 бит.
		Для типов со знаком отрицательные значения представлены с использованием дополнения до 2.
		Нет битов заполнения.

`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	Макс. 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
```