C LIBRARY GRIMOIRE

lomaster & oldteam

2023

Содержание

1 Введение **3**
1.1 Важные дополнения 3

2 assert.h **3**
2.1 Как использовать функцию assert() 5
2.2 Особенность заголовка assert.h 6

3 iso646.h **6**
3.1 Зачем это надо 7

4 stdlib.h **7**
4.1 Функции преобразование типов 8
4.2 Функции генерация псевдослучайных значений 9
4.3 Функции для работы с памятью 9
4.4 Функции для контроля выполнения программы 10

5 float.h **11**

6 stdbool.h **11**

7 limits.h **12**

1 Введение

Я все таки решился написать руководство по всей стандартной библиотеки C, но, прежде чем перейти к нему нужно понять что это вообще такое, и как же все начиналось.

А началось все в 1972 году, когда создатель Си (Dennis Ritchie), вместе с самим C, выкатывает и стандартную библиотеку для него. Сама библиотека, котороя состояла из довольно маленького набора функций, которые были прописаны еще и в самом компиляторе, уже содержала в себе функции для: Ввода-вывода, работы со строками, символами, математикой, и файлами. Также по casual UNIX традиции, открывать код компиляторов нельзя, поэтому мы так и не сможем увидеть код первой библиотеки, хотя наверно и к лучшему.

Дальше по лору в игру вступили ANSI C, которые в 1989 году выкатывают стандарт по стандартной библиотеки Си. И Международная организация по стандартам его спокойно подтверждает. Но они были не одни, попутно с ними работали и POSIX, которые хреначили уже свою спецификацию для стандартной библиотеки Си.

По итогу в наше время все сложилось так, что ANSI C задал самую базу, вроде имен заголовочных файлов, основных функций и другого, а POSIX разширил эту базу. Вообще я тут упустил довольно много, потому что стандарты Си выходят до сих пор, но это будет слишком долго и не особо интересно, тем более их до сих пор делают ANSI C и POSIX, вообщем главное я тут сказал.

1.1 Важные дополнения

Это руководство не по языку Си, а по последнему стандарту стандартной библиотеки, это (C11), и оно не привязано конкретно к ngulibc, тут будет и POSIX стандарт.

2 assert.h

<**assert.h**> - определяет макрос функцию **assert**, которую используют для проверки ошибок путем условия, например:

```
1 assert(size == 100);
```

Сам макрос определен на основе другой функции:

```
#define assert(e) \
((e) ? (void)0 : __assert(__func__, __FILE__, __LINE__, #
   e))
```

Может показатся сложным, но это совсем не так. Вот сама функция на которой он основан:

```
void
__assert(const char* func, const char* file, int line,
   const char* expression);
```

Все ее аргументы указывают местоположение вызова функции **assert**, которая вернула **false**. Первый ее аргумент это функция, второй файл, третий строчка, и четвертый это выражение которое вызвало ошибку, **aka** условие которое вернуло **false**.

Макрос тут нужен просто для упрощения использования этой функции, ведь пользователю будет просто лень, указывать файл, строчку, и т.д.

Давайте разберем макрос: **(e)** - это условие которое указывает пользователь, например: **10 == 9**. Затем с помощью тернарного оператора, идет проверка статуса этого условия, другими словами проверка, оно **true** или **flase**. Если оно **true**, тогда макрос просто ничего не делает, указание **(void)0** нужно только для компилятора, что бы он не выдавал предупреждения. Если же условие **false** тогда идет вызов функции **assert** которую мы до этого разбирали, вот как выглядит вызов:

```
__assert(__func__, __FILE__, __LINE__, #e))
```

Макрос **func** определяет компилятор, это название функции в которой находится вызов этого макроса, соответственно и вызов **assert**, со всеми остальными макросами это: **FILE**, **LINE**, аналогично. Можно и по названию догадаться.

Что за **хэштег e**? Если вы не знали то с помощью **хэштега** в макросах, можно застаить компилятор превратить аргумент этого макроса, в тип данных **string**.

Было:

```c
#define (e) e
/*
 * result: 10 == 9
*/
```

Стало:

```c
#define (e) # e
/*
 * result: "10 == 9"
*/
```

PS: Он просто взял значение аргумента **e** в кавычки.

2.1 Как использовать функцию assert()

В данном примере ничего не произойдет, потому что **10** действительно равно **10**-ти:

```c
/* main.c */
int main(void)
{
  assert(10 == 10); /* return true */
}
```

Вот другой пример в котором все иначе:

```c
/* main.c */
int main(void)
{
  assert(10 == 9); /* pipec */
}
```

В этом случае условие вернет **false**, поэтому ваша программа сразу же завершится, и в консоль вы получите такой вывод:

Assertion failed: (10 == 9), function main, file main.c, line 4.

Тут показано: само условие, функция, файл, и строчка, которая и вызывала эту ошибку. Этот текст функция запишет в поток **stderr**, это поток для ошибок.

PS: Программа завершается с кодом (**1**)

Также если функция **assert** вернула **false**, но это было не в функции, то вывод будет таким: **Assertion failed: (10 == 9), file main.c, line 4.**

2.2 Особенность заголовка assert.h

Особенность это макрос **NDEBUG**, определив который, перед включением <**assert.h**>, вы можете сделать так что бы все функции **assert** в вашем файле, просто пропускались, если точнее то всегда были **(void)0**.

```
1  /* main.c */
2  #define NDEBUG
3  #include <assert.h>
4
5  int main(void)
6  {
7    assert(10 == 9);
8  }
```

Таким образом, несмотря на то что условие вернуло **false**, функция **assert** просто ничего не будет делать.

3 iso646.h

<**iso646.h**> - определяет **11** макросов для высокоуровневой замены побитовых и логических операторов. Был добавлен в стандарте **C90** в **1995** году, скорее всего **1** апреля).

```
1  #define and     &&
2  #define and_eq  &=
3  #define bitand  &
4  #define bitor   |
5  #define compl   ~
6  #define not     !
7  #define not_eq  !=
8  #define or      ||
9  #define or_eq   |=
10 #define xor     ^
```

```
11 #define xor_eq  ^=
```

Это и есть все его макросы, думаю объяснять не нужно.

3.1 Зачем это надо

На самом деле у этого действительно был смысл, дело в том что раньше **QWERTY**-клавиатуры (как у нас сейчас), были не у всех. И расположение таких символов было просто адским, на некоторых клавиатурах. И легче было написать **and**, чем тянуть руку на метр до сраного амперсанда.

4 stdlib.h

<**stdlib.h**> - расшифровывается как "standard library". И содержит в себе функции для преобразованием типов, выделения памяти, генерации рандомных чисел, сортировки, поиска, математики, и контроля процесса выполнения программы.

Давайте вначале разберем макросы и новые типы данных, которые он определяет:

```
1 #define NULL ((void *) 0)
```

Это тот самый **NULL**, но такое определение он имеет именно в **C**, в C++ же его опредление выглядет иначе:

```
1 #define NULL nullptr
```

Но есть еще одно определение, которое работает только при компиляции через **gcc**:

```
1 #define NULL __null
```

В данном случае **gcc** определяет свой **NULL**.

Дальше идет определение нового типа данных, это **size t**, который используется для представления размера объекта. Также заметьте что его определение условное на основе макроса **WORDSIZE**, это размер машинного слова, **aka**, разрядность процессора.

```
#if __WORDSIZE == 64
  typedef ___uint64_t ___size_t;
#else
  typedef ___uint32_t ___size_t;
#endif
```

Но <**stdlib.h**> определяет еще **2** типа данных, это **div t** и **ldiv t**, которые используется только для функций **div** и **ldiv**, как их возвращаемое значение.

```
typedef struct
{
  int quot, rem;
} div_t;

typedef struct
{
  long int quot, rem;
} ldiv_t;
```

4.1 Функции преобразование типов

Большинство функций тут начинаются с «**ato**», это название происходит от «**ASCII to**» на русском (ASCII в). Другие же начинаются с «**strt**», это происходит от «**string to**» на русском (строка в).

atof - преобразование строки в **double**(число двойной точности), обратите внимание что для **float** она не подходит! *(C89, C89).*

atoi - одна из самых популярных функций, преобразование строки в **int**(целое число) *(C89, C99).*

atol - преобразование строки в **long int**(длинное целое число) *(C89, C99).*

atoll - преобразование строки в **long long int**(еще более длинное целое число) *(C99).*

strtod - преобразование строки в **double**, аналог **atof**, но предоставляет более подробную информацию об ошибке, в случае таковой *(C89, C89)*.

strtof - преобразование строки в **float**(число одиночной точности) *(C99)*.

strtol - преобразование строки в **long int**, также как и с **strtod** аналогична другой функции (**atol**), и отличается от таковой также более подробному учету ошибок *(C89, C99)*.

strtold - преобразование строки в **long double**(длинное число двойной точности) *(C99)*.

strtoll - преобразование строки в **long long int**, ситуация такая же как и с **strtol** и **strtod** *(C99)*.

strtoul - преобразование строки в **unsigned long int**(беззнаковое длинное целое число) *(C89, C99)*.

strtoull - преобразование строки в **unsigned long long int**(беззнаковое еще более длинное целое число) *(C99)*.

4.2 Функции генерация псевдослучайных значений

rand - генерирует псевдослучайное значение, основана на функции **randomr**, которая в большинстве библиотек **C**, использует алгоритм **LCG** (**Linear Congruential Generator**) для генерации *(C89, C99)*.

srand - устанавливает начальное значение(**seed**) для генератора псевдослучайных чисел *(C89, C99)*.

4.3 Функции для работы с памятью

malloc(Memory Allocation) - одна из самых популярных функций во всей библиотеки, используется для выделения блока памяти заданного размера в куче (от англ. heap), куча это область памяти, доступная для динамического выделения и освобождения *(C89, C99)*.

Примечание: Память, выделенная с помощью **malloc**, не инициализируется, и её содержимое остаётся неопределенным.

calloc(Contiguous Allocation) - также используется для выделения блока памяти в куче, но в отличие от **malloc**, она инициализирует всю выделенную память нулями. Это особенно полезно при работе с массивами или структурами, где необходимо гарантировать, что все элементы инициализированы *(C89, C99)*.

realloc(Reallocate Memory) - используется для изменения размера ранее выделенного блока памяти. Она принимает указатель на существующий блок памяти, новый размер, и, если возможно, перемещает данные в новый блок памяти указанного размера *(C89, C99)*.

Примечание: Оригинальный блок памяти освобождается.

free(Free Allocated Memory) - используется для освобождения выделенной в куче памяти. Освобождение памяти позволяет системе операционной системы переиспользовать этот участок памяти для других целей. После вызова **free**, указатель на эту память становится недействительным, и он больше не должен использоваться. Не освобожденная память может привести к утечкам памяти, что может вызвать проблемы в работе программы *(C89, C99)*.

4.4 Функции для контроля выполнения программы

Большинство функций здесь это системные вызовы, и написаны на **assembler-e**.

abort - используется для завершения программы с кодом ответа **SIGABRT** (6), применяется именно для завершения программы в случае ошибки.

atexit - как по мне это довльно по философски, используется для регистрации функции, которая будет вызвана перед завершением программы. Она не завершает программу сразу, а лишь указывает функцию которая будет выполнена перед завершение программы.

```
1 #include <stdio.h>
```

```
#include <stdlib.h>

void goodbye(void)
{
  printf("goodbye\n");
}

int main()
{
    atexit(goodbye);
    printf("Hello, world!\n");
    printf("Hello, world!\n");
    printf("Hello, world!\n");

    return 0;
}
```

Функции **goodbye** будет вызвана именно на **return 0**.

5 float.h

<**float.h**> - определяет макросы для ограничения параметров и типов данных с плавающей точкой. Вообщем просто аналог **limits.h**, но для типов данных **float** и **double**. *(C89, C89).*

6 stdbool.h

<**stdbool.h**> - определяет **4** макроса для работы с типом данных **bool**. Почему для работы с ним? Потому что сам тип данных **bool** определяет компилятор, ввиде такого макроса:

```
_Bool
```

Поэтому в файле **stdbool.h** тип данных **bool** определяется вот так:

```
#define bool _Bool
```

Затем идет определение двух значений для него:

```
#define true   1
#define false  0
```

Под конец файла, определяется макрос, который указывает что тип данных **bool** определен:

```
#ifndef __bool_true_false_are_defined
  #define __bool_true_false_are_defined  1
#endif
```

7 limits.h

<**limits.h**> - определяет макросы для ограничения целочисленных типов данных. Причем стоит учитывать, что размеры типов данных семейства **long**, завясят от разрядности процессора. Разрядность определяется с помощью константы **WORDSIZE**, если она равна **64**, то процессор **64-x** битный.