Обзор stdlibc.

stdio.h :

Поскольку си родился при создании ОС UNIX, то фундаментальное понятие при вводе/выводе это файл. Идея была такая, попробуем сделать работу со всеми устройствами как будто это реальный файл, пусть всё будет файлом. Экран - это файл, туда можно писать, клавиатура - это файл, с неё можно читать, есть реальные файлы – области на диске.

Проблема в функциях fprintf в том, что вывод буферизованный.

fflush(f); - вызвали и точно все записалось.

stdin – файл ассоциированный с вводом, stdout – с выводом в консоль, stderr – тоже, но он не буферизованный.

printf(“%d”,n) == fprintf(stdout,”%d”,n); это удобно использовать для отладки: было : FILE\* f=fopen(….);

fprintf(f,….) меняете f на stdout.

Двоичные файлы. Текстовые файлы предназначены для чтения человеком, хотя на самом деле всегда всё хранится в байтах. Это вопрос интерпретации. Некоторые символы ‘\n’ например переводят строку, ‘\t’ – табуляция. В Windows есть спец. символ EOF.

Двоичные файлы предназначены для чтения программой, соглашений по умолчанию особых там нет.

assert.h : главное что там есть – функция assert, используемая при отладке приложений, assert(s1!=null). Такого рода проверки нужны на этапе тестирования. Это достигается следующим: #define DEBUG, тогда assert будет работать, если нет, то это выражение преобразуется в пустую инструкцию “;”. Внутри assert стоит текст #ifdef DEBUG тогда инструкции else ; Постоянно убирать и вставлять #define DEBUG неудобно. У gcc есть ключ –D, например gcc –DDEBUG будет означать, что в файл добавили #define DEBUG. Вообще ошибки бывают двух видов: ошибки по вине программиста (может происходить всегда – на диске закончилось место, в памяти нет места, кабель отключен сетевой …) аварийно завершать программу не нужно, иногда можно продолжать работу с нормальной проверкой, просто вывести сообщение пользователю. То есть часто вместо assert нужны if и проверки. Вторая проблема, в release версии assert вообще не останется.

complex.h, ctype.h : if ( (‘A’<=c && c<=’Z’) || (‘a’<=c && c<=’Z’) ) проверка символа на букву, такого рода функции находятся в ctype.h аналог этого – функция isalpha (char c) вернёт 1, если c буква, 0 в противном случае. Есть ещё много проверок, на цифру, на букву в верхнем регистре, на знак препинания и подобные.

errno.h : есть стандартный путь принятый в Си для возвращения информации об ошибках. Если функция должна возвращать указатель, но не может, возвращает NULL, если должна число вернуть, но не может, возвращает -1. Но это не всегда работает, например функция тангенс - когда она вернёт ошибку не понятно. Есть переменная errno – целочисленная. Функции должны записывать число в эту переменную, если произошла ошибка. В errno.h содержатся макросы, расшифровывающие значения ошибок.

float.h : содержит константы связанные с переменными float и double, FLT\_MAX, DBL\_MAX, также FLT\_EPSILON, DBL\_EPSILON. double f()… double f2()… if( fabs (f()-f2()) < DBL\_EPSILON ) … это будет проверка того, что функции возвращают одинаковый результат в пределах погрешностей вычислений.

limits.h : константы INT\_MAX, UINT\_MAX, INT\_MIN.

inttypes.h : здесь написан набор typedef такого рода int8\_t , int32\_t.

math.h : fabs, cos, sqrt.

stdlib.h : функции вида int atoi (char\* str), float atof (char \*str), преобразование строки в число, в отличии от sscanf возвращает 0 (а если в строке был 0). Две функции srand(unsigned int seed), rand() – для вычисления псевдослучайных чисел. Идея такая: первое число – зерно (seed), есть некоторая формула, например Xn = Xn-1 \* Xn-1 (mod p) – все последующие числа вычисляются по формуле ряда. У Кнута написано, как выбрать формулу, чтобы последовательность удовлетворяла статистическим требованиям (типа выдавала случайные числа). Если в программе написать srand(113); то при двух запусках программы при вызове rand() будут всегда получаться одни и те же числа. Поскольку формула задана, она будет выдавать те же результаты при втором, третьем и т.д. запусках программы. Чтобы числа получались разные при разных запусках – можно привязаться ко времени. srand((unsigned int)time(NULL)); В UNIX time возвращает время в секунд с 01.01.1970, в Windows другая дата. Также в этом заголовочном файле находится функция exit(int code) – аварийного завершения программы. int system (char\* str); - запуск программы, например system (“rm \*.o”); по возвращаемому значению можно понять как отработала программа. Функции сортировки и поиска: qsort, bsearch – быстрый поиск в отсортированном массиве.

string.h : strchr, strlen, strcmp, strstr (char\* str1, char\* str2) – поиск подстроки str2 в str1, strtok( str, “ “) – разбиение строки на части : char \*p = strtok (str, “ “); while (p!=NULL) { p=strtok(NULL, “ “); }, скорее всего в реализации этой функции используются статические переменные.

time.h : time\_t , time\_t time (time\_t \*a) – можно вызвать a=time(NULL) либо time(&a); Есть функции переводящие число в строчку (день недели год и т.д.). clock() – возвращает время, которое потратил процессор с момента запуска вашей программы в тиках системных часов. Обычно используется для замера времени, поиска самых медленных участков вашего кода. Вообще для такого есть профилировщики, но можно на коленке самостоятельно сделать.

С коротким обзором libc всё, всего там 15-16 заголовочных файлов.