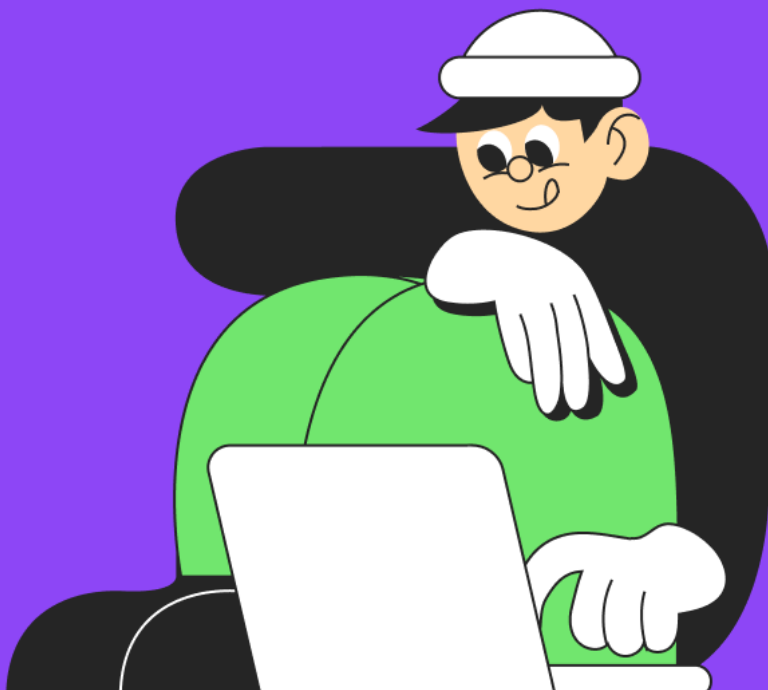
GeekBrains

Урок 11

Препроцессор

Аргументы командной строки

Программирование на языке Си (базовый уровень)



Оглавление

Введение

На предыдущей лекции вы узнали:

- Как разрабатывать и собирать многомодульные программы
- Как работать с файловой системой
- Подготовимся к курсовому проекту

На этой лекции вы узнаете:

-
-

Термины, используемые в лекции

Функция — фрагмент программы, описывающий решение некоторой подзадачи.

Аргументы командной строки

Одним из способов передачи данных в программу - передача их через аргументы командной строки при ее вызове.

```
ls -lt /etc
cat /etc/hosts
```

В данном примере мы вызываем утилиту ls и передаем ей два аргумента на вход. Во втором примере вызываем cat и передаем один аргумент - полное имя файла.

В Си программе, при запуске управление передается в функцию main. Эта функция получает на вход аргументы командной строки, которые можно в дальнейшем обработать.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    for(int i=0; i<argc; i++)
        printf("argc = %d, argv =
%s\n",i,argv[i]);
    return 0;
}
```

```
./prog hello world
argc = 0, argv = ./prog
argc = 1, argv = hello
argc = 2, argv = world
```

Первый аргумент функции main - **argc**, содержит количество аргументов переданных в программу. Второй аргумент - **argv** массив ссылок на отдельные слова переданных аргументов.

🔥 **Внимание!** Нулевой аргумент - это всегда название исполняемого файла самой программы.

Обработка аргументов getopt

В системе Linux существует два вида параметров: короткие и длинные. Короткие параметры начинаются с одного дефиса и имеют длину в один символ, их просто и быстро набирать в командной строке. Длинные параметры начинаются с двух дефисов и могут иметь длинное имя, которое целесообразно использовать в

скриптах (чтобы потом можно было вспомнить, что и как происходит). Кроме этого любой параметр может иметь значение, а может и не иметь.

```
-h          - короткий параметр
--help      - длинный параметр

-s 10       - параметры со значениями
--size 10
--size=10
```

Существует несколько специальных библиотек предназначенных для разбора списка переданных параметров:

- ***int getopt(...)*** - Обрабатывает короткие параметры
- ***int getopt_long(...)*** - Обрабатывает короткие и длинные параметры
- ***int getopt_long_only(...)*** - Обрабатывает параметры только как длинные

Разберемся с работой первой функции - ***getopt(...)***.

```
#include <unistd.h>

int getopt(int argc, char * const argv[],
    const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;
```

Эта функция последовательно перебирает переданные параметры в программу. Для работы в функцию передается количество параметров **argc**, массив параметров **argv**[] и специальная строка **optstring**, в которой перечисляются названия коротких параметров и признаки того, что параметры должны иметь значение. Например, если программа должна воспринимать три параметра **a, b, F** , то такая строка бы выглядела как "abF". Если параметр должен иметь значение, то после буквы параметра ставится двоеточие, например параметр **F** и **d** имеют значения, а параметры **e, a** и **b** не имеют, тогда эта строка могла бы выглядеть как "eF:ad:b". Если параметр может иметь (т.е. может и не иметь) значение, то тогда ставится два знака двоеточия, например "a::" (это специальное расширение GNU).
Для перебора параметров функцию **getopt**() надо вызывать в цикле. В качестве результата возвращается буква названия параметра, если же параметры кончились, то функция возвращает -1. Индекс текущего параметра хранится в **optind**, а значение параметра помещается в **optarg** (указатель просто указывает на элемент массива **argv**[]). Если функция находит параметр не перечисленный в списке, то выводится сообщение об ошибке в **stderr** и код ошибки сохраняется в **opterr**, при

этом в качестве значения возвращается "?". Вывод ошибки можно запретить, если установить **opterr** в 0.

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]){
    int rez=0;
//    opterr=0;
    while ( (rez = getopt(argc,argv,"ab:C::d")) != -1){
        switch (rez){
        case 'a': printf("found argument \"a\".\n"); break;
        case 'b': printf("found argument \"b = %s\".\n",optarg);
break;
        case 'C': printf("found argument \"C = %s\".\n",optarg);
break;
        case 'd': printf("found argument \"d\"\n"); break;
        case '?': printf("Error found !\n");break;
        };
    };
};
```

Скомпилируем и запустим программу

```
$ gcc -o prog main.c

$ ./prog -a -b -d -C hello
found argument "a".
found argument "b = -d".
found argument "C = hello".

$ ./prog -a -b -C -d
found argument "a".
found argument "b = -C".
found argument "d"

$ ./prog -a -b1 -C -d
found argument "a".
found argument "b = 1".
found argument "C = -d".

 ./prog -b1 -b2 -b 15
found argument "b = 1".
found argument "b = 2".
found argument "b = 15".
```

Посмотрим, как функция **getopt** вылавливает ошибки. Попробуем задать параметр, которого нет в списке:

```
$ ./prog -h -a
./prog: illegal option -- h
Error found !
found argument "a".
```

Можно отключить вывод сообщений об ошибках, для этого надо где-то в программе перед вызовом функции вставить **opterr=0**;.

```
$ gcc -o prog main.c
$ ./prog -h -a
Error found !
found argument "a".
```

Пример работы с файловой системой Аргументы командной строки

```
#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>
enum {PATH_LENGTH=256};

#define STR255 "%255s"

void convert_path_to_full(char *full_path, const char *dir) {
    if(dir[0]=='/') {
        strcpy(full_path, dir);
    } else if (dir[0]=='.') {
        getcwd(full_path,PATH_LENGTH);
    }
    else {
        getcwd(full_path,PATH_LENGTH);
        strcat(full_path, "/");
        strcat(full_path, dir);
    }
    if(full_path[strlen(full_path)-1] !='/')
```

```
        strcat(full_path,"/");// добавляем / в конце
}
void print_filetype(int type) {
    switch (type) {
        case DT_BLK:  printf("b "); break;
        case DT_CHR:  printf("c "); break;
        case DT_DIR:  printf("d "); break; //directory
        case DT_FIFO:  printf("p "); break; //fifo
        case DT_LNK:  printf("l "); break; //Sym link
        case DT_SOCK: printf("s "); break; //Filetype isn't
identified
        default:       printf("  "); break;
    }
}
/**
 Расширить  строку пробелами.
 @print_lenth длинна до которой надо расширить
 */
void print_space(int print_lenth, int str_lenth) {
    while( (print_lenth - str_lenth)>0 ) {
        putchar(' ');
        str_lenth++;
    }
}
void print_tab(int tab_number) {
    for(int t=1; t<tab_number; t++) {
        putchar('\t');
    }
}
void print_file_size(long long int byte_number) {
    if(byte_number/(1024*1024))
        printf("%lld Mb", byte_number/(1024*1024));
    else if(byte_number/1024)
        printf("%lld Kb", byte_number/1024);
    else
        printf("%lld b", byte_number);
}
void ls(const char *dir) {
    static int tab_count = 0; //уровень вложенности рекурсии
    tab_count++;
    struct stat file_stats;
    DIR *folder;
    struct dirent *entry;
    int files_number = 0;
    char full_path[PATH_LENGTH]={0};
```

```
    convert_path_to_full(full_path, dir);
    folder = opendir(full_path);

    if(folder == NULL)
    {
        perror("Unable to read directory ");
        printf("%s\n",full_path);
        return;
    }

    while( (entry=readdir(folder)) )
    {
        if( entry->d_name[0]=='.' )// пропускаем поддиректории
            continue;
        char full_filename[PATH_LENGTH]={0};
        files_number++;
        print_tab(tab_count);//отступы при рекурсии
        printf("%4d : ",files_number);
        print_filetype(entry->d_type);

        strcpy(full_filename, full_path);
        strcat(full_filename, entry->d_name);

        printf("%s", entry->d_name);
        print_space(20, entry->d_namlen);
        if (!stat(full_filename, &file_stats))
        {
            print_file_size(file_stats.st_size);
            printf("\n");
        }
        else
        {
            printf("stat failed for this file\n");
            perror(0);
        }
    }
    closedir(folder);
    tab_count--;
}

int main(int argc, char *argv[])
{
    char dir[PATH_LENGTH], buf[PATH_LENGTH];
    int rez=0;
```

```
//    opterr=0;
    while ( (rez = getopt(argc,argv,"hf:")) != -1){
        switch (rez){
            case 'h': printf("This is example of list
directory\n");
                printf("Usage: clear [options]\n\
                 -h This help text\n\
                 -f Specify folder.\n");
                printf("Example: %s -f /tmp\n",argv[0]);
                return 0;
        case 'f': //printf("folder is \"f = %s\".\n",optarg);
                strcpy(dir, optarg);
                break;
        case '?': printf("Unknown argument: %s Try -h for
help\n", argv[optind-1]);
                return 1;
        };
    };

    convert_path_to_full(buf, dir);
    //printf("ls for folder %s\n",buf);
    ls(dir);
    return 0;
}
```

Что такое debugging

Итак, что же такое дебаггинг? От английского слова bug - поиск жуков :-)

Отладка - это процесс, в котором разработчик, используя различные доступные ему инструменты, пытается изменить часть ранее написанного кода, чтобы избавиться от ошибок, которые не были обнаружены во время компиляции.

Существует несколько способов отладки:

- **Трассировка**. Вывод сигнальных сообщений. Используем макросы для debug вывода. Отключение части кода и анализ. Требуется некоторое время для того чтобы написать все операторы печати.

```
#define DEBUG
#ifdef DEBUG
    #define D if(1)
#else
    #define D if(0)
```

```
#endif

D printf("This is debug message\n");
```

- **Логирование**. Этот метод также основан на тексте, как и трассировка, но с множеством полезных свойств. Вместо использования только «stderr» в качестве механизма вывода, можно отправлять сообщения на специальные системы логирования. Это очень эффективный способ. А ведение журнала является обязательным условием для любого серьезного производственного приложения. Но у него также есть свои ограничения. Нам по-прежнему приходится вручную писать все операторы журнала внутри нашего кода, а также мы не можем заглядывать под капот выполнения программы.
  https://github.com/christianhujer/sclog4c
  http://log4c.sourceforge.net/

```
//Логирование вручную
#include <stdio.h>

int main(int argc, char *argv[])
{
    fprintf(stderr,"Debug
message\n");
    fprintf(stdout,"Stdout
message\n");
    return 0;
}
```

```
:$ ./prog
Debug message
Stdout message
:$ ./prog 2>./error_log.txt
Stdout message
:$ cat error_log.txt
Debug message
```

```
//Логирование sclog4c
#include <stdio.h>
#include "sclog4c.c"

int main(int argc, char *argv[])
{
  sclog4c_level = SL4C_ALL;
  logm(SL4C_DEBUG, "Program name: %s",
argv[0]);
  return 0;
}
```

```
:$ ./prog
main.c:8: debug: In
function main: Program
name: ./prog
:$ ./prog
2>./error_log.txt
:$ cat error_log.txt
main.c:8: debug: In
function main: Program
name: ./prog
```

- Отладка с использованием **отладчика**. Преимущество такого рода отладки состоит в том, что с вашей стороны не требуется никаких дополнительных усилий для начала отладки кода. Вы просто запускаете отладчик и указываете его на исполняемый файл, и вуаля! вы находитесь в середине работающей программы, но у вас также есть все возможности исследовать (и даже при необходимости изменить) ее в реальном времени.

Препроцессор

Препроцессор Си обеспечивает текстовую обработку файла программы до передачи ее компилятору. Результатом работы препроцессора является текстовый файл, который далее попадает на вход основной стадии трансляции.

Каждая директива препроцессора должна быть записана в отдельной строке файла. При необходимости директива препроцессора может быть продолжена на следующую строку с помощью символа ‘\’. Отличительным признаком директивы препроцессора является символ #, который должен быть первым непробельным символом в строке.

Задачи препроцессора:

- обеспечение модульности программы
- обеспечение переносимости программы (за счет условной компиляции)
- автоматическая генерация однотипных участков кода (за счет макросов)

Вообще говоря, до этапа препроцессирования компилятор также производит предварительную подготовку текста программы – обрабатывает многобайтовые символы, заменяет комментарии пробелом и т.п.

```c
#include <stdio.h>

#define DEBUGPRINT fprintf
(stderr, "debug in %d line\n",
__LINE__)

int main(void)
{
    DEBUGPRINT;
    return 0;
}
```

```
debug in 7 line
```

Если в ходе макроподстановки получившаяся строка снова содержит известные препроцессору имена макросов, то к этим именам рекурсивно применяется макроподстановка до получения строки, не содержащей имена макросов.

```
#include <stdio.h>

#define ONE printf ("DEBUG\n")
#define TWO ONE; ONE
#define FOUR TWO; TWO

int main(void)
{
    FOUR;
    return 0;
}
```

```
DEBUG
DEBUG
DEBUG
DEBUG
```

Рассмотрим более сложный пример макроса

```
    #define A B + 1
    #define B 2

    int a;
    a = A + 2;
    printf("a = %d\n", a);
```

```
a = 5
```

Определение макроса с параметрами выглядит следующим образом:

```
/* *
 Открывающая скобка не должна быть отделена пробельными символами
от имени макроса. Если в списке параметров или в тексте
встречаются комментарии, каждый комментарий заменяется на один
символ пробела.
 */
#define SWAP(a,b) (a ^= b, b ^= a, a ^= b)
int main(void)
{
    int a=1,b=2;
    printf("%d %d\n",a,b);
    SWAP(a,b);
    printf("%d %d\n",a,b);
    return 0;
}
```

```
    #define HALF(x) x/2 //так
неправильно

    int a=5, b;
    b = HALF(a + 5);
    printf("a = %d b = %d\n",a,
b);
```

```
a = 5 b = 7
```

```
/ *
Для расстановки приоритетов операций
необходимо заключать в скобки как тело
макроса, так и все вхождения в него
формальных аргументов
*/
    #define HALF(x) ((x)/2)
    int a=5, b;
    b = HALF(a + 5);
    printf("a = %d b = %d\n",a, b);
```

```
a = 5 b = 5
```

```
#define SUM(a,b) (a) + (b)
    int a=5, b;
    b = SUM(a,3) * 10 ;
    printf("a = %d b = %d\n",a,
b);
```

```
a = 5 b = 35
```

```
#define SUM(a,b) ((a) + (b))
    int a=5, b;
    b = SUM(a,3) * 10 ;
    printf("a = %d b = %d\n",a,
b);
```

```
a = 5 b = 80
```

Макросы, которые используют свои аргументы несколько раз, могут порождать ошибки связанные с побочным эффектом.

```
#define MAX(a,b)
((a)>=(b)?(a):(b))

    int a=5, b=7, c;
    c = MAX(a,++b) ;
    printf("c = %d\n",  c);
```

```
 c = 9
```

```
#define SWAP(type,a,b) {type t = a; a =
b; b = t;}

    int a=5, b=7;
    if(a<b)
        SWAP(int, a,b);
    else
        a = 1000;
printf("a = %d b = %d\n", a, b);
```

```
error: expected
expression
```

```
#define SWAP(type,a,b) do{type t = a; a =
b; b = t;}while(0)

    int a=5, b=7;
    if(a<b)
        SWAP(int, a,b);
    else
        a = 1000;
printf("a = %d b = %d\n", a, b);
```

```
a = 7 b = 5
```

Вопрос. Если в качестве одного из аргументов будет указана переменная t? Как поправить?

```
//Преобразование аргумента в строку
#define TOSTR(a) #a
printf("%s\n",TOSTR(hello world));
printf("%s\n",TOSTR(123));
printf("%s\n",TOSTR("hello world"));
```

```
hello world
123
“hello world”
```

```
//Операция склейки записывается как <arg1>##<arg2>
    #define MERGE(a,b) a##b
    MERGE(d,o) // do
    MERGE(a,2) // a2
    MERGE(+,+) // ++
```

```
#define ANAME(n) a##n
#define  PRINT_AN(n)  printf("a"
#n " = %d\n", a ## n);
```

```
    int ANAME(1) = 10;
    int ANAME(2) = 22;
    int ANAME(3) = 35;

    PRINT_AN(1)
    PRINT_AN(2)
    PRINT_AN(3)

    int i=3;
    PRINT_AN(i)

    MERGE(a1+,+);
    PRINT_AN(1)
```

```
a1 = 10
a2 = 22
a3 = 35


error:    use    of    undeclared
identifier 'ai'

a1 = 11
```

Предопределенные макросы

```
#include <stdio.h>
void why_me();

int main(void)
{
    printf("File: %s\n", __FILE__);
    printf("Date: %s\n", __DATE__);
    printf("Time: %s\n", __TIME__);
    printf("Version: %ld\n",
__STDC_VERSION__);
    printf("Line: %d\n", __LINE__);
    printf("Function: %s\n",
__func__);
    why_me();
    return 0;
}
void why_me() {
    printf("Function: %s\n",
__func__);
    printf("Line: %d\n", __LINE__);
}
```

```
$ gcc -o prog main.c
-std=c99
:$ ./prog
File: main.c
Date: Mar 11 2021
Time: 16:03:27
Version: 199901
Line: 10
Function: main
Function: why_me
Line: 17
:$ gcc -o prog main.c
-std=c11
:$ ./prog
File: main.c
Date: Mar 11 2021
Time: 16:03:32
Version: 201112
Line: 10
Function: main
Function: why_me
Line: 17
```

Условная компиляция

Препроцессор также позволяет организовать включение определенных частей файла в окончательный текст файла в зависимости от некоторых условий. Существуют следующие директивы: if, #ifdef или #ifndef, они заканчиваются #endif, внутри может быть #elif. Общий принцип работы такой же как и в if-else. Эти

директивы позволяют проверить, определено ли конкретное имя, и в зависимости от этого включить или не включить текст между #ifdef и #endif в окончательный текст файла программы.

Основное назначение директив условной компиляции - задавать фрагменты программы, которые должны или не должны компилироваться в зависимости от значения некоторого макроса. Например, программа может компилироваться в двух режимах: отладочном и рабочем. Отладочный режим может обозначаться макросом DEBUG. Тогда мы можем определить макрос для отладочной печати, который в отладочном режиме будет раскрываться в некоторый оператор, выводящий отладочную печать, а в рабочем режиме — в пустую строку.

```
#include <stdio.h>

#define DEBUG

#if defined DEBUG
    #define DEBUGPRINT fprintf (stderr, "debug in %d line func:
%s\n", __LINE__, __func__)
#else
    #define DEBUGPRINT
#endif

int main(void)
{
    int a=5, b=7;
    DEBUGPRINT;
    printf("a = %d b = %d\n", a, b);
    return 0;
}
```

Другим важным случаем использования условной компиляции является защита от двойного включения заголовочного .h файла в .c файл реализации. Заголовочные файлы содержат объявления переменных и функций, которые могут привести к ошибкам в случае их неоднократного включения в текст программы. Для большой программы со сложными связями между отдельными заголовочными файлами ситуация двойного включения одного файла, возможно, через другие заголовочные файлы, может легко возникнуть. Во избежание такой ошибки обычно используют следующий прием: для каждого заголовочного файла определяют уникальное имя макроса, и весь текст файла подключается лишь при условии, что данное имя не определено. Например, для стандартного файла stdio.h:

```
#ifndef _STDIO_H
```

```
#define _STDIO_H
<... текст файла ...>
#endif
```

Также, условная компиляция может использоваться для комментирования больших фрагментов кода. Комментарии в языке Си не могут вкладываться друг в друга, поэтому невозможно закомментировать текст функции с помощью /* и */, если он уже содержит такие комментарии. Тогда нужно использовать условную компиляцию:

```
#if defined COMMENT_THIS
<some code to comment>
#endif
```

Условная компиляция — для фрагментов программ, которые выглядят по-разному в разных операционных системах.

```
#if defined __APPLE__
    <code for mac os>
#elif defined __linux__
      <code for Linux>
#endif
```

Макросы с динамическим числом аргументов

```
//  Вместо ... подставляется __VA_ARGS__

#define PR(X,...) printf("Message " #X ": "
"__VA_ARGS__)

int main(void)
{
1.
    return 0;
}
```

```
Message 2: Hello
10
```

Директивы #line и #error

```
#line 100 "help.c"

#ifndef __APPLE__
#error This code work only in MacOs
#endif
```

```
Текущий номер строки 100 а файл help.c


При компиляции не на MacOs будет ошибка
```


Что же выбрать, макрос или функцию?

Задачи можно решать и с помощью макроса и с помощью функций. Нет строгих правил что лучше. Можно дать лишь несколько рекомендаций, которые позволят принять правильное решение.

Если макрос вызывать 20 раз, то в программу вставляются 20 строк этого макроса. Если функцию вызвать 20 раз, то в программе содержится единственный экземпляр ее операторов, но программа совершает переход к фрагменту кода функции и обратно, что в целом замедляет ее работу, т.е. Этот процесс может занимать больше времени чем выполнение встраиваемого кода.

Для макроса не важны типы и размеры, он манипулирует со строками.

Стоит выбрать макрос вместо функции если:

- Если макрос можно описать одной строкой
- Можно использовать макрос для реализации простых функций
- Если вы намерены использовать макрос вместо функции для ускорения, стоит подумать, а даст ли это желаемый эффект. Воспользуйтесь **профилированием**, для определения скорости работы программы

Подведение итогов

Итак, на этой лекции мы начали с изучения функций их определение и вызовов. Зачем нужен прототип или описание функции, как передавать параметры в функцию и получать результат. Рассмотрели вопросы область видимости и модификатор static.

Изучили особенности буферного и не буферного ввода и как работают потоки ввода и вывода.

Познакомились с ASCII-таблицей и ее свойствами.

Рассмотрели функции ввода-вывода getchar() и putchar(), а также функции не буферизированного ввода getch() и getche().

Затронули тему оформление функций.

Традиционно продолжили написание программы вычисления корней квадратного уравнения в части доработки систему меню и добавляя функций, а также произвели рефакторинг, кода. Еще раз на практике рассмотрели модификатор static переменных.

Познакомились с машиной состояний и конечными автоматами и реализовали конечный автомат машины для варки кофе.

Дополнительные материалы

1. Оформление программного кода http://www.stolyarov.info/books/pdf/codestyle2.pdf
2. Стандарт разработки программного обеспечения MISRA на языке C http://easyelectronics.ru/files/Book/misra_c_rus.pdf
3. https://ru.wikipedia.org/wiki/ASCII#%D0%9D%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82%D1%8B_ASCII ASCII
4. https://habr.com/ru/company/ruvds/blog/548672/ Решайтесь на великие поступки — ASCII
5. https://blog.skillfactory.ru/glossary/ascii/ ASCII
6. https://www.ap-impulse.com/shag-30-ds1307-i-avr-dvoichno-desyatichnyj-format-bcd-vyvodim-vremya-na-indikator/ Шаг №30. DS1307 и AVR. Двоично-десятичный формат BCD
7. https://habr.com/ru/companies/timeweb/articles/717628/ С чем едят конечный автомат
8. https://tproger.ru/translations/finite-state-machines-theory-and-implementation/ Конечный автомат: теория и реализация
9. https://metanit.com/c/tutorial/7.7.php Консольный ввод-вывод
10. https://cpp.com.ru/kr_cbook/ch7kr.html Глава 7. Ввод и вывод

Используемые источники

1. cprogramming.com - учебники по C и C++

2. free-programming-books - ресурс содержащий множество книг и статей по программированию, в том числе по C и C++ (там же можно найти ссылку на распространяемую бесплатно автором html-версию книги Eckel B. «Thinking in C++»)
3. tutorialspoint.com - ещё один ресурс с множеством руководств по изучению различных языков и технологий, в том числе содержит учебники по C
4. Юричев Д. «Заметки о языке программирования Си/Си++» - «для тех, кто хочет освежить свои знания по Си/Си++»
5. Онлайн версия «Си для встраиваемых систем» Функции
6. https://metanit.com/c/tutorial/4.1.php Функции