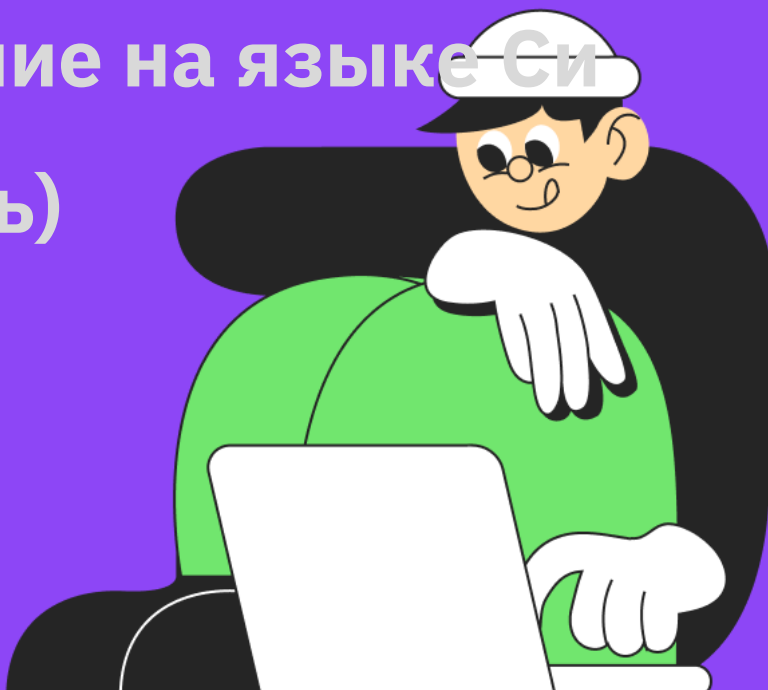
GeekBrains

Урок 10

Многомодульные программы

Динамические структуры данных

Программирование на языке Си (базовый уровень)



Оглавление

Введение

На предыдущей лекции вы:

- Как работать с файловой системой
- Файлы текстовые
- Бинарные файлы
- Множество сканирования scanf и fscanf
- Узнаем, что такое структуры, битовые поля и объединения и как с ними работать

На этой лекции вы узнаете:

- Как разрабатывать и собирать многомодульные программы
- Как работать с файловой системой

- Подготовимся к курсовому проекту

Термины, используемые в лекции

Компилятор — получает из каждой единицы трансляции код на машинном языке (Ассемблер) и генерирует объектный модуль с машинным кодом.

Препроцессор — компонент, производящий набор текстовых подстановок над файлом для получения его окончательного вида и передачи компилятору. Подстановка текстов заголовочных файлов (директива #include), условная трансляция, макроподстановки.

Линковщик — обеспечивает слияние нескольких объектных файлов в один исполняемый.

Классы памяти

При объявлении переменных можно указать класс памяти в котором данная переменная будет размещена компилятором в последствии. Это определит ее область видимости и время жизни. В языке Си определены 4 класса памяти: **extern**, **static**, **auto**, **register**. Мы уже рассматривали их на Уроке 3.

- **auto** — переменная создана при входе в блок и уничтожена при выходе из него. Область видимости такой переменной - блок в котором она объявлена.
- **register** — может использоваться при определении формальных параметров функций и переменных внутри блока. Это рекомендация транслятору, попытаться разместить данную переменную на регистре, а не в памяти.
- **static** — переменная с таким классом будет существовать все время работы программы. Область видимости ограничена блоком в котором данная переменная объявлена. Глобальная переменная с таким классом будет доступна только из файла в котором она объявлена.
- **extern** — будет означать, что память под переменную будет выделена в другом файле. Переменная будет существовать все время работы программы.

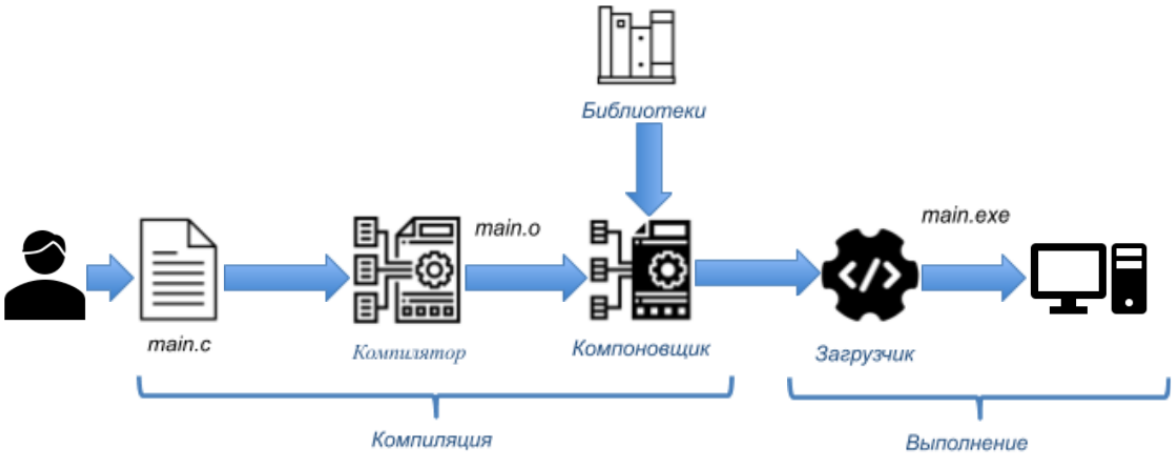
Многомодульные программы

В реальной жизни программы состоят не из одного файла, а из нескольких. Эти файлы могут быть заголовочными (с расширением .h), обычно они содержат описание функций с помощью которых можно использовать данную программу и

файлы содержащие реализацию отдельных частей самой программы. В языке Си можно компилировать каждый файл отдельно, а потом объединить откомпилированные файлы в одну программу.

В ходе **компиляции**[1] одного файла сначала этот файл обрабатывается **препроцессором**[2]. Потом компилятор получает ассемблерный код, то есть представляет программу в виде последовательности команд процессора и описания необходимых ячеек в глобальной и статической памяти. Далее ассемблер получает по файлу на языке ассемблера (это объектный файл, имеющий расширение .o), в котором команды закодированы в двоичном виде, а также описан вид статической и глобальной памяти. В конце **линковщик**[3] из нескольких объектных файлов создает исполняемый файл программы.

Все переменные (кроме блочных) и функции в Си по умолчанию видны из всех файлов. Можно использовать ключевое слово *global*, чтобы дополнительно подчеркнуть это. Если же к переменной или функции добавить ключевое слово *static*, то это сделает переменную или функцию видимой только внутри файла где она описана.



Рассмотрим пример. В файле ***main.c*** текст основной программы из которой происходит вызов функции max. Сама функция находится в другом файле - ***func.c***. Необходимо будет собрать два исходных файла в один исполняемый файл ***prog***.

main.c	func.c
`#include <stdio.h>`	`int max(int a, int b)` `{`

[1] **Компилятор** - Получает из каждой единицы трансляции код на машинном языке (Ассемблер) и генерирует объектный модуль с машинным кодом.
[2] **Препроцессор** - компонент, производящий набор текстовых подстановок над файлом для получения его окончательного вида и передачи компилятору. Подстановка текстов заголовочных файлов (директива #include), условная трансляция, макроподстановки
[3] **Линковщик** - обеспечивает слияние нескольких объектных файлов в один исполняемый.

```c
extern int max(int, int); // Функция описана в др файле

int m; // Глобальная переменная. Видна из обоих файлов

static int sm; // Глобальная переменная. Видна только из main.c

int main(void) {
    int a,b;
    scanf("%d%d",&a,&b);
    m = max(a,b);
    printf("max(%d %d) = %d\n",a,b,m);
    return 0;
}
```

```c
    return a>b? a : b;
}
```

Пример сборки программы из двух модулей.

gcc -c -o main.o main.c	Команда создает объектный файл из исходного
gcc -c -o func.o func.c	Команда создает объектный файл из исходного
gcc -o prog func.o main.o	Линковка двух объектных файлов в исполняемый

Вынесем объявление функции max() из примера в отдельный заголовочный файл *mylibrary.h*.

main.c

```c
#include <stdio.h>
#include "mylibrary.h"

int m;

int main(void) {
    int a,b;
    scanf("%d%d",&a,&b);
    m = max(a,b);
    printf("max(%d %d) = %d\n",a,b,m);
    return 0;
```

mylibrary.h

```c
int max(int, int);
```

func.c

```c
#include "mylibrary.h"
int max(int a, int b) {
    return a>b? a : b;
}
```

`}`	

Вопросы к примеру:

1. Зачем в файле func.c подключается заголовок?
2. Можно ли пометить функцию max как extern? Как static?
3. Какие команды для сборки необходимо использовать?

🔥 **Внимание!** Может понадобиться добавить к команде компиляции ключ -I. для того, чтобы компилятор нашел локальный заголовочный файл.

Рассмотрим еще пример. Добавим функцию нахождения максимума из трех чисел.

`main.c`

```
#include <stdio.h>
#include "mylibrary.h"

int m;

int main(void) {
    int a,b,c;
    scanf("%d%d%d",&a,&b,&c);
    m = max_3(a,b,c);
    printf("max(%d %d %d) = %d\n",
```

`mylibrary.h`

```
// Функция доступна только из func.c
static int max(int, int);

// Функция доступна везде
int max_3(int, int, int);
```

`func.c`

```
#include "mylibrary.h"
```

```
a, b, c, m);
    return 0;
}
```

```
int max_3(int a, int b, int c)
{
    return max(max(a,b),c);
}

static int max(int a, int b) {
    return a > b ? a : b;
}
```

Подключение библиотек

Для линковки с дополнительными библиотеками используются опции -l и -L. Опция -l используется для указания имени библиотеки, с которой будет производиться линковка. При этом префикс lib отбрасывается.

Пример линковки с математической библиотекой libm:

```
gcc -lm -o program file1.o file2.o
```

Если библиотека расположена в нестандартном месте (стандартное место зависит от платформы), то необходимо указать путь к ней опцией -L.

Пример линковки с указанием пути к библиотекам:

```
gcc -L/usr/local/lib -lm -o program file1.o file2.o
```

Утилита make

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки. Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.

Пример: make [-f make-файл] [цель] ...

Если опция -f не указана, используется имя по умолчанию для make-файла — Makefile.

make открывает make-файл, считывает правила и выполняет команды, необходимые для создания указанной цели.

Стандартные цели для сборки дистрибутивов GNU:

- **all** — выполнить компиляцию пакета (цель по умолчанию)
- **install** — установить пакет из дистрибутива (производит копирование исполняемых файлов, библиотек и документации в системные директории)
- **uninstall** — удалить пакет (производит удаление исполняемых файлов и библиотек из системных директорий)
- **clean** — очистить дистрибутив (удалить из дистрибутива объектные и исполняемые файлы созданные в процессе компиляции)
- **distclean** — очистить все созданные при компиляции файлы и все вспомогательные файлы созданные утилитой ./configure в процессе настройки параметров компиляции дистрибутива

Программа make выполняет команды согласно правилам, указанным в специальном файле. Этот обычный текстовый файл с именем - *Makefile*. Как правило, он описывает, каким образом нужно компилировать и компоновать(линковать) программу.

make-файл состоит из правил и переменных. Правила имеют следующий синтаксис:

```
цель1 цель2 ...: реквизит1 реквизит2 …
<-TAB->команда1
<-TAB->команда2
         ...
```

Правило представляет собой набор команд, выполнение которых приведёт к сборке файлов-целей из файлов-реквизита.

Правило сообщает make, что файлы, получаемые в результате работы команд (цели) являются зависимыми от соответствующих файлов-реквизита. make никак не проверяет и не использует содержимое файлов-реквизита, однако, указание списка файлов-реквизита требуется только для того, чтобы make убедилась в наличии этих

файлов перед началом выполнения команд и для отслеживания зависимостей между файлами.

Обычно цель представляет собой имя файла, который генерируется в результате работы указанных команд. Целью также может служить название некоторого действия, которое будет выполнено в результате выполнения команд (например, цель clean в make-файлах для компиляции программ обычно удаляет все файлы, созданные в процессе компиляции).

💡 **Важно!** Строки, в которых записаны команды, должны начинаться с символа табуляции.

Составим Makefile для предыдущего примера

```
 Makefile
all: prog

prog: main.o func.o
	gcc -o prog main.o func.o

main.o: main.c mylibrary.h
	gcc  -c -o main.o main.c

func.o: func.c
	gcc  -c -o func.o func.c

clean:
	rm -rf *.o prog
```

Если в процессе разработки программы в файл mylibrary.h будут внесены изменения, потребуется перекомпиляция обоих файлов и линковка, а если изменим func.c, то повторную компиляцию main.o можно не выполнять.

make - утилита облегчающая процесс сборки многомодульных программ.

Makefile - текстовый файл содержащий набор правил для утилиты make.

Задачи про датчик (модули)

Делаем отдельные модули

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#define SIZE 30
struct sensor {
    uint8_t day;
    uint8_t month;
    uint16_t year;
    int8_t t;
};
void cgangeIJ(struct sensor* info,int
i,int j)
{
struct sensor temp;
    temp=info[i];
    info[i]=info[j];
    info[j]=temp;
}
//упорядочивающую его по неубыванию
температуры
void SortByT(struct sensor* info,int
n)
{
    for(int i=0; i<n; ++i)
        for(int j=i; j<n; ++j)
            if(info[i].t>=info[j].t)
                cgangeIJ(info,i,j);
}
```

```c
unsigned int DateToInt(struct
sensor* info)
{
    return info->year << 16 |
info->month << 8 |
           info->day;
}
//упорядочивающую его по дате
void SortByDate(struct sensor*
info,int n)
{
    for(int i=0; i<n; ++i)
        for(int j=i; j<n; ++j)

if(DateToInt(info+i)>=
               DateToInt(info+j))

cgangeIJ(info,i,j);
}
void AddRecord(struct sensor*
info,int number,
uint16_t year,uint8_t
month,uint8_t day,int8_t t)
{
    info[number].year = year;
    info[number].month = month;
    info[number].day = day;
    info[number].t = t;
}
```

Вызываем модули

```
int AddInfo(struct sensor* info)
{
int counter=0;

AddRecord(info,counter++,2021,9,16,9);
AddRecord(info,counter++,2022,9,2,-9);
AddRecord(info,counter++,2021,1,7,8);
AddRecord(info,counter++,2021,9,5,1);
    return counter;
}

void print(struct sensor* info,int
number)
{
printf("================================
=====\n");
    for(int i=0;i<number;i++)
        printf("%04d-%02d-%02d
t=%3d\n",
            info[i].year,
            info[i].month,
            info[i].day,
            info[i].t
        );
}
```

```
int main(void)
{
struct sensor info[SIZE];
int number=AddInfo(info);
    print(info,number);
    printf("\nSort by t\n");
    SortByT(info,number);
    print(info,number);
    printf("\nSort by date\n");
    SortByDate(info,number);
    print(info,number);

    return 0;
}
```

Вычисление корней квадратного уравнения (модули)

Вызов модуля модуль

Давайте сделаем систему меню на основе не буферизированного ввода.

```
#include <stdio.h>
#include <conio.h>
#include <locale.h>
int main(int argc, char **argv)
{
char Choice;
    setlocale(LC_ALL, "Rus");
    while(1)
```

```
    {
        printf("1. Вычисление корней квадратного уравнения\n");
        printf("0. Выход\n");
        printf("Для выход нажмите Q\n");
        Choice = getch();
        switch(Choice)
        {
            case '1':
                SquareEquation();
            break;
            case '0':
            case 'q':
            case 'Q':
                return 0;
            break;
            default:
                printf("Непонятный выбор %x\n",Choice);
            break;
        }
    }
    return 0;
}
```

Делаем отдельные модули

Сделаем вычисление корней квадратного уравнения отдельной функцией.

```
void SquareEquation(void)
{
float a,b,c;
float B,d;
float X1,X2;
    printf("Вычисление корней квадратного уравнения \\
\"a*x*x+b*x+c=0\"\n");
    printf("Введите a:\n");
    scanf ("%f", &a); //1
    printf("Введите b:\n");
    scanf ("%f", &b); //18
    printf("Введите c:\n");
    scanf ("%f", &c); //32
    B = b/2;
    if(a!=0)
    {
        d = B*B-a*c;
        if(d<0)
        {
            printf("Корни квадратного уравнения комплексные \n");
        }
        else
        {
            printf("Корни квадратного уравнения \n");
```

```
            d = sqrtf(d);
            X1 = (-B + d)/a; //2
            printf("X1 = %f \n",X1);
            X2 = (-B - d)/a; //16
            printf("X2 = %f \n",X2);
        }
    }
    else
    {
        if(b!=0)
        {
            X1 = -c/b;
            printf("Корень линейного уравнения %f\n",X1);
        }
        else
        {
            printf("Корней НЕТ!\n");
        }
    }
}
```

Конечный вариант программы

```
#include <stdio.h>
#include <conio.h>
#include <locale.h>
#include <math.h>

float InputFloat(char* message)
{
float number;
static int counter = 0;
    counter++;
    printf("%d,%s",counter,message);
    scanf("%f",&number);
    return number;
}

void CalcRealRoots(float sqrD,float B,float a)
{
float X1,X2;
            printf("Корни квадратного уравнения \n");
            float d = sqrtf(sqrD);
            X1 = (-B + d)/a; //2
            printf("X1 = %f \n",X1);
            X2 = (-B - d)/a; //16
            printf("X2 = %f \n",X2);
}

//Сделаем вычисление корней квадратного уравнения отдельной
функцией.
```

```
void SquareEquation(void)
{
    printf("Вычисление корней квадратного уравнения \\
\"a*x*x+b*x+c=0\"\n");
    float a = InputFloat("Введите a:\n");//1
    float b = InputFloat("Введите b:\n");//18
    float c = InputFloat("Введите c:\n");//32
    float B = b/2;
    if(a!=0)
    {
        float d = B*B-a*c;
        if(d<0)
        {
            printf("Корни квадратного уравнения комплексные \n");
        }
        else
        {
            CalcRealRoots(d,B,a);
        }
    }
    else
    {
        if(b!=0)
        {
            float X1 = -c/b;
            printf("Корень линейного уравнения %f\n",X1);
        }
        else
        {
            printf("Корней НЕТ!\n");
        }
    }
}
int main(int argc, char **argv)
{
char Choice;
    setlocale(LC_ALL, "Rus");
    while(1)
    {
        printf("1. Вычисление корней квадратного уравнения\n");
        printf("0. Выход\n");
        printf("Для выход нажмите Q\n");
        Choice = getch();
        switch(Choice)
        {
            case '1':
                SquareEquation();
            break;
            case '0':
            case 'q':
            case 'Q':
                return 0;
```

```
            break;
            default:
                printf("Непонятный выбор %x\n",Choice);
            break;
        }
    }
    return 0;
}
```

Стек - LIFO и очередь - FIFO

Стек (англ. stack) – структура данных, предоставляющая доступ к элементам в порядке, обратном порядку добавления. Элемент, добавленный в стек последним, будет возвращен первой же операцией извлечения.

- Операци:
  push - добавить в стек принято.
- **pop** - извлечь из стека.

В очереди элементы извлекаются в том же порядке, в котором они были добавлены. Операции:

- **enqueue** - добавить в очередь.
- **dequeue** - взять из очереди.

```
// Пример организации стека как однонаправленного списка
#include <stdio.h>
#include <stdlib.h>
typedef int datatype;
typedef struct list {
    datatype value;
    struct list * next;
}stack;
void push(stack **p,datatype data)
{
    stack *ptmp;
    ptmp=malloc(sizeof(stack));
    ptmp->value=data;
    ptmp->next=*p;
    *p=ptmp;
}
_Bool empty_stack(stack *p) {
    return p==NULL;
}
datatype pop(stack **p)
```

```
{
    stack *ptmp=*p;
    datatype c;
    if(empty_stack(*p))
        exit (1); // Попытка взять из пустого стека
    c=ptmp->value;
    *p=ptmp->next;
    free(ptmp);
    return c;
}
int main()
{
    stack *p=NULL; // Важно для корректной работы присвоить NULL
    for(int i=1;i<=5; i++)
        push(&p,i);
    for(int i=1;i<=5; i++)
        printf("%d\n",pop(&p)); // 5 4 3 2 1
    return 0;
}
```

```
// Пример организации стека через динамический массив
#include <stdio.h>
#include <stdlib.h>
typedef int datatype;
typedef struct stack{
    datatype *item;
    int size;
    int sp;
} stack;

void init_stack(stack *st)
{
    st->size=16;
    st->sp=0;
    st->item=malloc(16*sizeof(datatype));
}

void delete_stack(stack *st)
{
    free(st->item);
}
void push (stack *st, datatype value)
{
```

```
    if(st->sp==st->size-1)
    {
        st->size=st->size*2;
        st->item=realloc(st->item,st->size*sizeof(datatype));
    }
    st->item[st->sp++]=value;
}

void pop (stack *st, datatype *value)
{
    if(st->sp < 1) {
        printf("stack empty");
        exit(1);
    }
    *value=st->item[--(st->sp)];
}
int empty_stack(stack *st)
{
    return (st->sp < 1);
}

int main()
{
    stack st;

    int a,i;
    init_stack(&st);
    do
    {
        scanf("%d",&a);
        push(&st,a);
    }while(a!=0);
    for(i=0;i<st.sp;i++)
        printf("%d ",st.item[i]);

    while(!empty_stack(&st))
    {
        pop(&st,&a);
        printf("%d ",a);
    }

    return 0;
}
```

```c
// Пример реализации очереди при помощи однонаправленного списка
#include <stdio.h>
#include <stdlib.h>
typedef int datatype;
typedef struct list {
    datatype value;
    struct list * next;
}queue;

_Bool empty_queue(queue *p) {
    return p==NULL;
}

datatype dequeue(queue **p)
{
    queue *ptmp=*p;
    datatype c;
    if(empty_queue(*p)) {// Попытка взять из пустой очереди
        fprintf(stderr,"Error. Queue is empty\n");
        exit(1);
    }
    c=ptmp->value;
    *p=ptmp->next;
    free(ptmp);
    return c;
}

void enqueue(struct list **pl, datatype data) {
    struct list  *ptmp = *pl, *elem;
    elem = malloc(sizeof(struct list));
    elem->value = data;
    elem->next = NULL;
    if(*pl==NULL) {
        *pl = elem;
        return;
    }
    while(ptmp->next)
        ptmp=ptmp->next;
    ptmp->next = elem;
}

void enqueue_recurs(struct list **pl, datatype data) {
    if(*pl == NULL) {
        (*pl) = malloc(sizeof(struct list));
        (*pl)->value = data;
```

```
        (*pl)->next = NULL;
        return;
    } else {
        enqueue_recurs(&((*pl)->next) ,data);
    }
}
void print_list(struct list *pl) {
    while(pl) {
        printf("%d ",pl->value);
        pl = pl->next;
    }
    printf("\n");
}
int main()
{
    queue *pq=NULL;

    for(int i=1;i<=5; i++)
        enqueue(&pq,i);
    for(int i=1;i<=5; i++)
        printf("%d\n",dequeue(&pq));

    return 0;
}
```

Пример работы с файловой системой

```
#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 длина 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(void)
{
    char dir[PATH_LENGTH], buf[PATH_LENGTH];
    printf("Input dir: ");
    scanf(STR255,dir);
    convert_path_to_full(buf, dir);
    printf("ls for folder %s\n",buf);
    ls(dir);
    return 0;
}
```

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