

# Лекция 9

Локальные и глобальные переменные, классы памяти  
Функция **main**: параметры и возвращаемое значение  
Функции с неопределенным количеством параметров  
Указатели на функции, массивы указателей на функции  
Рекурсия и рекурсивные функции

---

# Основные понятия

- **Локальное объявление** – объявление типа или переменной, находящееся внутри какой-либо пользовательской функции программы.
  - **Глобальное объявление** – объявление типа или переменной, находящееся вне какой-либо пользовательской функции программы.
  - **Локальная переменная** – переменная, объявленная внутри какой-либо пользовательской функции программы.
  - **Глобальная переменная** – переменная, объявленная вне какой-либо пользовательской функции программы.
-

---

# Пример

Программа запрашивает у пользователя размер целочисленного массива (ограничен – максимально 30 элементов). Затем осуществляется ввод массива и поиск максимального и минимального элементов массива. Найденные значения выводятся на экран. Данная программа должна быть реализована с использованием функционального подхода: необходимо реализовать функции ввода массива и поиска максимума и минимума массива.

---

# Пример

```
int array[30], num;
void Input(void);
void GetMinMax(int *, int *);
int main(int argc, char *argv[])
{
    printf("Введите количество элементов: ");
    scanf("%d",&num);
    if((num<1)||&num>30) {
        puts("Некорректный ввод!");
        return 0;
    }
    Input(); //Ввод массива
    int max, min;
    GetMinMax(&max,&min);
    printf("Максимум: %d\nМинимум: %d\n",max,min);
    return 0;
}
```

# Пример

```
void Input(void)
{
    puts("Введите массив.");
    for(int i=0;i<num;i++) scanf("%d",&array[i]);
}
```

```
void GetMinMax(int *max, int *min)
{
    *max = array[0], *min = array[0];
    for(int i=1;i<num;i++){
        if(*max < array[i]) *max = array[i];
        if(*min > array[i]) *min = array[i];
    }
}
```

---

# Локальные переменные

В языке C допускается (**но не рекомендуется**) объявление локальной переменной с тем же именем, что и глобальная переменная.

В таком случае в функции, где объявлена такая переменная, используется локальная переменная, а не глобальная. Например:

```
int value = 10;
void func(void);
int main(int argc, char *argv[])
{
    printf("%d\n",value);
    func();
    printf("%d\n",value);
    return 0;
}
void func(void)
{
    int value = 5;
    printf("%d\n",value);
}
```

---

---

# ОСНОВНЫЕ ПОНЯТИЯ

- **Время жизни** – промежуток времени в течении которого под переменную выделена память, следовательно она содержит некоторое значение и к ней можно обращаться.
  - **Видимость переменной** – область программы, в которой к данной переменной можно обращаться, т.е. переменная «видна» в этой области.
-

# Пример

Функция поиска в списке точек одной точки (структура POINT с двумя вещественными полями), расположенной наиболее близко центру координат.

```
POINT FindNearest(POINT list[],int num)
{
    POINT pnt = list[0];
    double min = sqrt(pow(pnt.x,2.0) + pow(pnt.y,2.0));
    for(int i=0;i<num;i++){
        double dist = sqrt(pow(list[i].x,2.0) +pow(list[i].y,2.0));
        if(dist < min) {min = dist; pnt = list[i];}
    }
    return pnt;
}
```



---

# Классы памяти

В языке C существует возможность управления временем жизни переменных. Для этого используются классы памяти, которые определяют некоторую специфику переменной. Для использования определенного класса памяти переменную необходимо объявить с указанием ее класса памяти:

класс памяти тип имя [=инициализация];

В языке C имеются четыре класса памяти:

- автоматический (**auto**);
  - регистровый (**register**);
  - статический (**static**);
  - внешний (**extern**).
-

---

# Класс `auto`

Класс памяти **`auto`** используется для создания локальных переменных в функциях. Переменная создается в момент вызова функции в стеке и уничтожается при ее завершении.

Данный класс памяти используется по умолчанию, при объявлении переменных его использовать необязательно.

---

---

# Класс `register`

Класс памяти **`register`** используется для создания целочисленных (или производных от целочисленного типа) переменных в регистрах процессора с целью ускорения доступа к ним.

Обычно с таким классом объявляют индексные переменные, используемые в циклах:

```
for(register int i=0;i<n;i++) ...
```



---

# Класс **static**

Класс памяти **static** используется для создания статических переменных.

Данный класс памяти используется по умолчанию при описании глобальных переменных. Переменные с таким классом памяти создаются в сегменте данных программы и «живут» в процессе выполнения программы.

Данный класс памяти можно использовать и для локальных переменных. В таком случае значение этой переменной не теряется между вызовами функции, в которой она описана.

---

---

# Пример

```
void Show(void)  
{  
    static int value = 0;  
    printf(“%d\n”,value);  
    value++;  
}  
  
int main(int argc, char *argv[])  
{  
    for(int i=0;i<5;i++) Show();  
    return 0;  
}
```

---

---

# Класс **extern**

Класс памяти **extern** используется для описания «внешних» переменных. Под внешней переменной здесь понимается переменная, которая будет описана где-то далее в программе. Таким образом, класс памяти **extern** используется как бы для описания ссылок на переменные.

---

---

# Функция `main`

Согласно стандарту описания функции `main` в нее могут передаваться параметры, и она может возвращать целочисленное значение.

Заголовок такой функции имеет вид:

```
int main(int argc, char *argv[])
```



---

# Функция `main`

При запуске программы в нее из операционной системы или другой программы могут быть переданы параметры командной строки.

**Командная строка** – строка, содержащая имя запускаемой программы (абсолютный путь к файлу программы) и следующие за ним параметры, представляющие собой некоторые символьные данные. Разделение имени программы и ее параметров осуществляется пробелами (одним или несколькими).

Например:

**`c:\programs\proga.exe first second`**

---



---

# Функция `main`

Параметры командной строки представлены в функции `main` двумя ее параметрами:

- целочисленным значением (обычно называемым `argc`);
- массивом строк (обычно называемым `argv`).

Пример:

- `argc == 3`
  - `argv == {"c:\programs\proga.exe", "first", "second"}`
-

---

# Функция `main`

Функция `main` может возвращать целочисленное значение, которое может интерпретироваться операционной системой или вызвавшей программой как результат выполнения данной программы (код ошибки).

Принято следующее правило: если программа выполнилась корректно, то ее результат должен быть равен нулю.

---

---

# Функция `main`

Пример: в программу в качестве параметров командной строки передаются целые числа. Программа должна вычислить сумму этих чисел и вернуть полученное значение.

```
int main(int argc, char *argv[])  
{  
    if(argc < 2) return 0;  
    int summa = 0;  
    for(int i=1;i<argc;i++) summa += atoi(argv[i]);  
    return summa;  
}
```

---

---

# Пример 1

Написать программу, вычисляющую сумму цифр целых чисел. Числа передаются в параметрах командной строки. Программа выводит информацию в формате: *число – сумма цифр*. Вычисление суммы цифр одного числа реализовать в виде функции.

---

# Пример 1

```
#include <stdio.h>
#include <stdlib.h>
unsigned Calc(int);
int main(int argc, char *argv[])
{
    for(int i=1;i<argc;i++){
        int num = atoi(argv[i]);
        unsigned summa = Calc (num);
        printf("%d - %u\n",num,summa);
    }
    return 0;
}
```

```
unsigned Calc(int num)
{
    unsigned res = 0;
    while(num >0){
        res += num%10;
        num /= 10;
    }
    return res;
}
```

## Пример 2

Написать программу подбора пароля к rar-архиву, если известно, что пароль состоит из трех цифр. Имя архива передается в параметрах командной строки. При реализации программы предполагается, что программа-распаковщик rar-архивов (unrar.exe) находится в том же каталоге, что и сама программа. Для вызова программы-распаковщика используется функция из библиотеки **stdlib.h**:

```
int system(const char *command);
```

# Пример 2

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  if(argc < 2) {puts("Не указано имя архива!"); return 0;}
  for(int i=0;i<10;i++)
    for(int j=0;j<10;j++)
      for(int k=0;k<10;k++){
        char psw[5] = "   ";
        psw[0] = i+48; psw[1] = j+48; psw[2] = k+48;
        char cmd[50] = "unrar.exe e -p";
        strcat(cmd,psw); strcat(cmd,argv[1]);
        int res = system(cmd);
        if(!res){printf("Пароль: %s\n",psw); return 0; }
      }
  puts("Пароль не найден!");
  return 0;
}
```

---

# Неопределенное число параметров

В языке C допускается создание функций, имеющих неопределенное число параметров. Для реализации функции с неопределенным числом параметров необходимо описать ее заголовок (и прототип, если он есть) в следующем виде:

**тип имя(список фиксированных параметров, ...)**

---



---

# Неопределенное число параметров

Функция описывается в обычной форме, только после указания последнего фиксированного параметра через запятую указываются три точки.

Например:

```
int function(int n, ...)
```

```
double func(int i, double val, ...)
```



---

# Неопределенное число параметров

Для доступа к значениям нефиксированных параметров используются типы и макросы для работы со стеком из библиотеки **stdarg.h**.

Работа по извлечению значений нефиксированных параметров заключается в следующем:

1. объявить переменную типа **va\_list**;
  2. установить ее на последний фиксированный параметр в функции с помощью макроса **va\_start**;
  3. произвести работу с заданным списком значений, используя макрос **va\_arg**;
  4. завершить работу со списком параметров, используя макрос **va\_end**.
-

---

# Неопределенное число параметров

Макрос установки переменной для работы со стеком на первый нефиксированный параметр:

```
va_start(va_list ap, lastfix);
```

Первым параметром *ap* макроса является имя переменной для работы со стеком, вторым параметром *lastfix* – имя последнего фиксированного параметра.

---

---

# Неопределенное число параметров

Макрос для получения значения следующего нефиксированного параметра:

```
va_arg(va_list ap, type);
```

Первый параметр ***ap*** макроса – имя переменной для работы со стеком. Второй параметр ***type*** макроса – имя типа получаемого значения (системный или пользовательский тип данных).

---

---

# Неопределенное число параметров

Макрос для завершения работы со стеком:

```
va_end(va_list ap);
```

В качестве параметра **ap** передается имя переменной для работы со стеком.

---

---

# Пример

Реализовать функцию, вычисляющую среднее арифметическое значение нескольких чисел. В функцию сначала передается количество значений (тип **int**), а затем сами значения (тип **double**).

---

# Пример

```
double Mid(int N,...)
{
  va_list ap;
  int i = 0;
  double S=0;
  va_start(ap,N);
  while(i<N){
    S+=va_arg(ap,double);
    i++;
  }
  va_end(ap);
  S/=N;
  return S;
}
```

Вызов приведенной в примере функции для вычисления среднего значения трех вещественных чисел (1, 2 и 3) будет иметь вид:

```
... Mid(3,1,2,3);
```

а пяти вещественных чисел (1, 2, 3, 4 и 5):

```
... Mid(5,1.0,2.0,3.0,4.0,5.0);
```

---

# Указатели на функции

В языке C можно объявлять указатели не только на данные, но и на функции.

Синтаксис объявления такого указателя следующий:  
тип (\*имя)(список типов формальных параметров);

---



---

# Указатели на функции

Например, если функция в качестве параметра принимает два целых числа, а возвращает вещественное значение, то указатель на эту функцию будет описан в виде:

```
double (*ptr)(int,int);
```

Указатель на функцию, принимающую в качестве параметра строку и целое число, и возвращающую строку будет иметь вид:

```
char *(*ptr)(const char *,int);
```

---

---

# Указатели на функции

Установка указателя на функцию осуществляется простым присвоением указателю имени функции:

указатель = имя функции;

Вызов функции, через установленный на нее указатель, осуществляется так же, как и обычный вызов функции:

указатель(список фактических параметров);

---

# Пример

Вывести на экран таблицу значений на промежутке  $[0, 2\pi]$  с шагом  $\pi/6$  одной из функции  $(1 - \sin, 2 - \cos)$ . Номер функции задан в целочисленной переменной num. Фрагмент программы:

```
double (*ptr)(double);  
if(num == 1) ptr = sin;  
    else if(num == 2) ptr = cos;  
    else return 0;  
const double pi = 3.14159265358979323;  
double x = 0.0;  
while(x <= 2.0*pi){  
    printf("func(%lf) = %lf\n",x,ptr(x));  
    x += (pi/6.0);  
}
```

---

# Массивы указателей на функции

Еще одной конструкцией языка С, встречающейся на практике, является массив указателей на функции. Объявление такого массива, на примере одномерного массива, имеет вид:

тип (\*имя[размер])(список типов формальных параметров);

Установка указателя на функцию осуществляется простым присвоением элементу массива имени функции:

массив[индекс] = имя функции;

Вызов функции, через установленный на нее элемент массива указателей, осуществляется так же, как и обычный вызов функции:

массив[индекс](список фактических параметров);

---

# Пример

Вывести на экран таблицу значений на промежутке  $[0, 2\pi]$  с шагом  $\pi/6$  одной из функции (1 – **sin**, 2 – **cos**, 3 – **tan**). Номер функции задан в целочисленной переменной num. Фрагмент программы:

```
double (*ptrs[3])(double);  
ptrs[0] = sin; ptrs[1] = cos; ptrs[2] = tan;  
if((num < 1)|| (num > 3)) return 0;  
const double pi = 3.14159265358979323;  
double x = 0.0;  
while(x <= 2.0*pi){  
    printf("func(%lf) = %lf\n", x, ptrs[num-1](x));  
    x += (pi/6.0);  
}
```

---

# Рекурсия

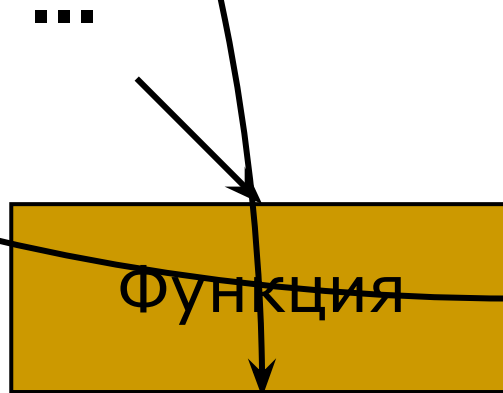
Функции в языке C могут использоваться рекурсивно. **Рекурсия** – вызов функции самой себя.

Различают два вида рекурсии:

- прямая рекурсия;
  - косвенная рекурсия.
-

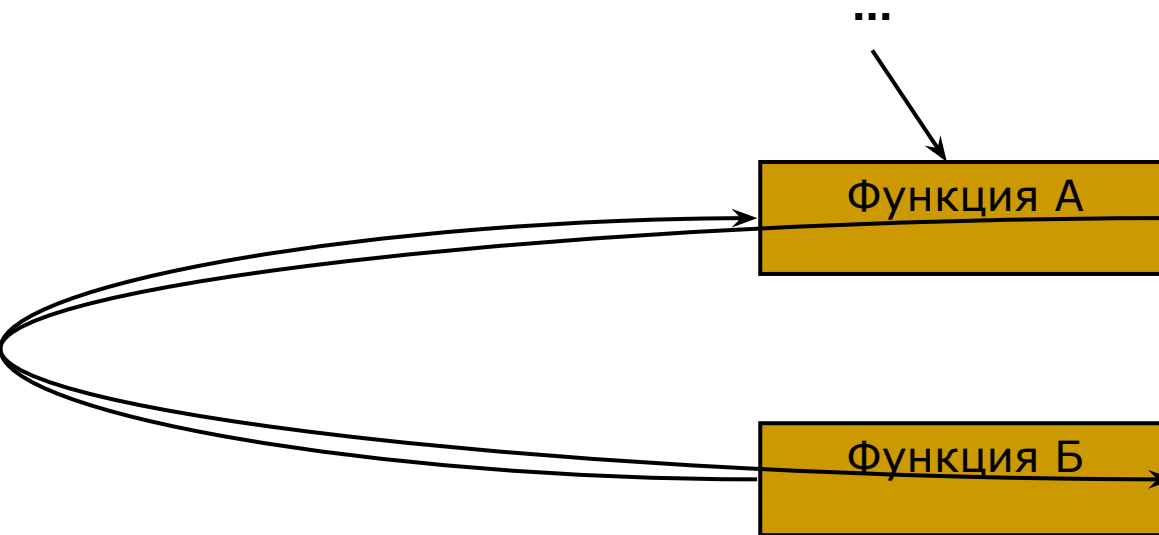
# Рекурсия

**Прямая рекурсия** – функция вызывает непосредственно саму себя. Изображение прямой рекурсии на функциональной схеме программы приведено на рисунке.



# Рекурсия

**Косвенная рекурсия** – функция вызывает себя посредством другой функции. Например, функция А вызывает функцию Б, которая, в свою очередь, вызывает функцию А. Изображение косвенной рекурсии на функциональной схеме программы приведено на рисунке.





# Пример

В качестве примера прямой рекурсии рассмотрим рекурсивную функцию вычисления факториала:

```
double Factorial(unsigned n)
{
    if(n == 1) return 1.0;
    return (double)n*Factorial(n-1);
}
```

# Пример 1

Написать программу поиска различных расстановок восьми ферзей на шахматной доске так, чтобы они не «били» друг друга. Программа должна отобразить на экране матрицу  $8 \times 8$ , состоящую из нулей (пустая клетка на шахматной доске) и единиц (ферзь на шахматной доске), а также общее число найденных комбинаций. При реализации использовать рекурсию.

```
--x-----      {2,5,3,1,7,4,6,0}  
-----x--  
---x-----  
-x-----  
-----x  
----x---  
-----x-  
x-----
```

# Пример 1

```
#include <stdio.h>  
void PutFerz(int,int [], int *);  
int main(int argc, char *argv[])  
{  
    int l[8] = {0}, Count = 0;  
    PutFerz(0,l,&Count);  
    printf("Количество комбинаций: %d\n",Count);  
    return 0;  
}  
  
int Check(int n, int l[])  
{  
    for(int i=0;i<n;i++){  
        if(l[i]==l[n]) return 1;  
        if(abs(n-i)==abs(l[n]-l[i])) return 1;  
    }  
    return 0;  
}
```

# Пример 1

```
void Show(int l[], int *count)
{
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++)
            printf("%d",i==l[j]);
        printf("\n");
    }
    puts("=====");
    (*count)++;
}
```

```
void PutFerz(int n, int l[], int *count)
{
    for(l[n]=0;l[n]<8;l[n]++){
        if(Check(n,l)) continue;
        if(n==7) Show(l,count);
        else PutFerz(n+1,l,count);
    }
}
```

# Пример 1

