ЛАБОРАТОРНАЯ РАБОТА № 10
ФУНКЦИИ

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

При программировании различных задач часто приходится использовать одни и те же алгоритмы (например, сортировка массива, определение максимального элемента, отбражение графика, определение корня уравнения, перемножение матриц и т. д.). Для того чтобы каждый раз не составлять программы, реализующие подобные алгоритмы, целесообразно оформить их в виде подпрограмм или, как их называют в языке С, функций. Такие функции можно использовать в дальнейшем и в независимых друг от друга программах, и в одной программе несколько раз.

Функция — это именованная последовательность описаний и операторов (инструкций), выполняющая какое-либо законченное действие. Функция может принимать параметры (иметь аргументы) и возвращать значение. Примером может быть функция, вычисляющая значение синуса числа:

```
y = sin(x);
```

Здесь **sin** – имя функции, **x** – аргумент, передаваемый в функцию. Функция вычисляет значение синуса, которое и присвивается переменной **y**.

Деление программы на функции является основным принципом структурного программирования. Преимущества использования функций:

- при использовании функций упрощается структура программы;
- использование функций позволяет избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы;
- часто используемые функции можно помещать в библиотеки и затем применять их в других программах;
- упрощается процесс отладки программ.

Любая программа на языке С состоит из функций, одна из которых должна иметь имя **main** (или **WinMain**), с нее начинается выполнение программы. Во время работы из фунции **main** могут вызываться другие функции, из них третьи и т.д.

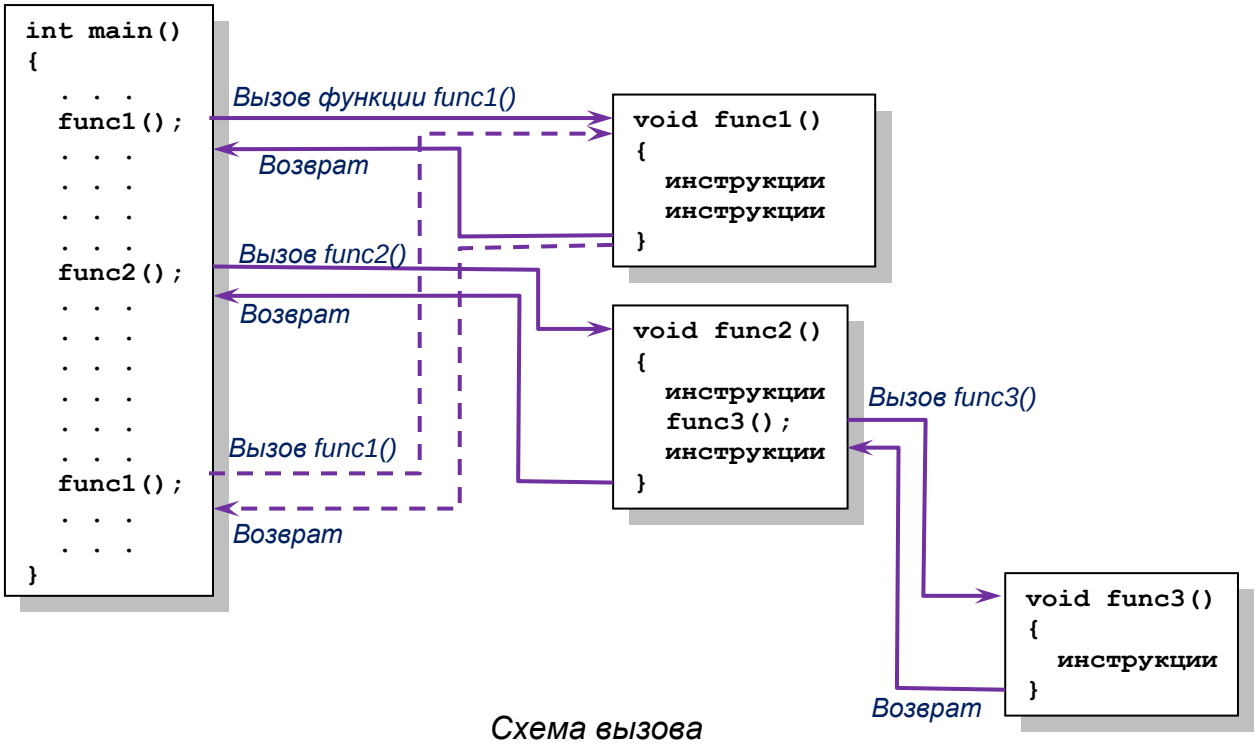


Схема вызова

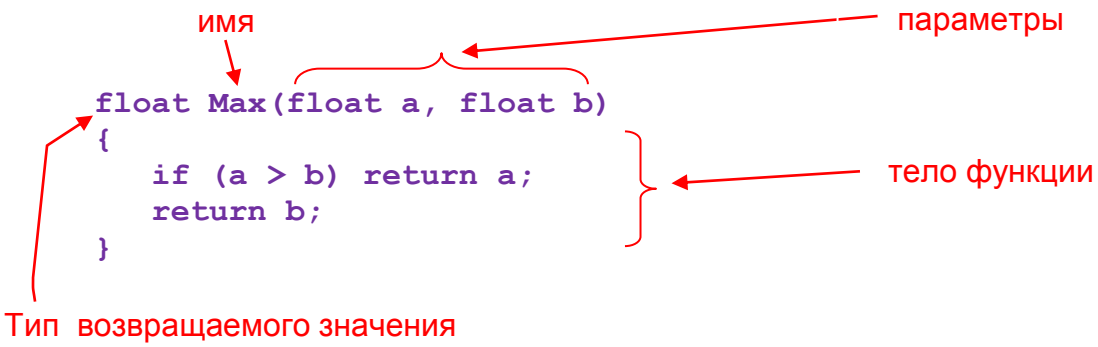
Обращение к функции осуществляется посредством ее вызова, при этом обязательно указывается имя функции. Кроме того в функцию могут передаваться параметры, необходимые для ее выполнения. Функция начинает выполняться в момент вызова. По завершении функции управление передается следующей после вызова инструкции.

Для того, чтобы функцию можно было использовать в прогремме, ее сначала необходимо создать (т.е. **объявить и определить**).

Определение функции состоит из **заголовка** и тела **функции**. В заголовке указывается **тип** возвращаемого значения, **имя** и список передаваемых **параметров**. **Тело функции**, представляет собой последовательность инструкций и описаний в фигурных скобках.

```
Тип_возвращаемого_значения   Имя_функции   (список_параметров)
{
    инструкции // тело функции
}
```

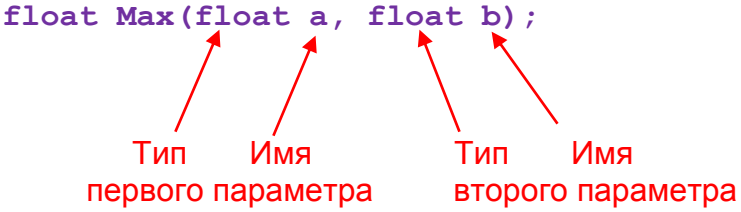
Например, функция, возвращающая максимальное из двух значений:



Тип возвращаемого функцией значения может быть любым, кроме массива (но может быть указателем на массив). Если функция не должна возвращать никакого значения, **вместо типа возвращаемого значения** пишется слово **void** (пустой). Например, функция, которая выводит на дисплей надпись, заданную строкой в параметрах функции, и ждет нажатия клавиши, не должна ничего возвращать:

```
void ViewStr(const char* str)
{
   printf("%s\n", str);
   getch();
}
```

Функция может иметь один или несколько параметров или не иметь их вовсе. Для каждого параметра, передаваемого в функцию, указывается его тип и имя. Эти параметры становятся локальными переменными функции.



Если параметров у функции нет, пишется слово **void** или скобки оставляют пустыми, например:

```
int getch();
```

Объявление функции (или предварительное объявление или **прототип**) должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог

осуществить проверку правильности вызова. Объявление функции состоит из заголовка функции, после которого стоит точка с запятой.

```
float Max(float a, float b);
```

Прототипы функций обычно помещают в заголовочные файлы (с расширением **h**), а определения – в файлы с расширением **c** или **cpp**). Например, функция, вычисляющая синус, объявляется в файле **math.h** примерно так:

```
double sin(double x);
```

Необходимо помнить, что в определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать.

Каждая функция должна выполнять **одну точно поставленную задачу** и ее **имя** должно **соответствовать смыслу** этой задачи.

Количество и типы параметров, а также тип возвращаемого значения зависят от конкретной задачи, выполняемой функцией. Например, у функции, определяющей максимальное значение из двух чисел, должно быть два параметра. Тип параметров определяется тем, какие числа мы собираемся сравнивать, если вещественные, то **float**. Функция должна возвращать максимальное из этих чисел, следовательно, тип возвращаемого значения должен быть тоже **float**.

В прототипе функции можно задавать только типы параметров (имена не обязательны), но при определении функции нужно указать не только типы параметров, но обязательно их имена. Они становятся именами локальных переменных функции, значения этих переменных задаются при каждом вызове функции.

Для **вызова функции** необходимо написать ее имя, а если функция имеет параметры, то в скобках подставить значения аргументов, например:

```
float f1, f2, fmax;
f1 = 10.5;
f2 = 22.3;
fmax = Max(f1, f2); // Вызов функции Max
```

> Параметры, перечисленные в заголовке определения функции, называются формальными параметрами, или просто **параметрами**, а записанные в операторе вызова функции — фактическими параметрами или **аргументами**.

Аргументы в функцию передаются по значению или через указатели (а в языке C++ еще по ссылке). Для передачи значений аргументов используется специальная область памяти, которая называется **стек** (см. приложение). В данном примере (функция **Max**) аргументы передаются по значению, т.е. значение переменной **f1** копируется в локальную переменную **a** данной функции, а значение переменной **f2** – в локальную переменную **b.**

Если аргументы в функцию передаются по значению, их исходные значения не изменяются.

В качестве аргументов в функцию можно передавать и массивы, и строки. При передаче массива в качестве параметра нужно в заголовке функции после формального имени массива указать пустые квадратные скобки.

```
float MaxInArray(float Array[], int Len); // прототип
```

В функцию передается массив переменных типа **float**. Чтобы функция знала размер массива, необходимо передать ей этот размер, например с помощью еще одной переменной, назовет ее **Len**.

Передать массив в функцию можно также в форме указателя:

```
float MaxInArray(float* Array, int Len); // прототип
```

Сам массив при передаче в качестве параметра в функцию не копируется, в функцию передается лишь указатель на начало этого массива, это значит, что элементы массива в функции можно изменить

При передаче в функцию строки (массива символов) размер не нужен, т.к. конец строки можно определить по коду **'\0'**.

```
int MStrLen(const char* s);                    // прототип
```

При вызове функции имя массива-аргумента задается уже без квадратных скобок. Пример вызова данных функций:

```
float arr[10];
float f = MaxInArray(arr, 10);  // вызов
. . .
int i = MStrLen("ABCDE");    // вызов
```

Кроме параметров в самой функции можно объявлять и другие **локальные переменные**.

Ранее уже говорилось, что переменные, которые используются в программах, можно разделить на две группы: **локальные** и **глобальные**. Локальные переменные – это переменные, которые определяются внутри функции или внутри блока программы (блок – это набор директив, заключенный в фигурные скобки), и используются только внутри функции (блока).

Локальные переменные, объявленные в таком блоке, создаются при входе в него и ликвидируются при выходе. Переменные, которые объявлены в одном блоке, не имеют никакого отношения к переменным другого блока даже в случае совпадения имен.

Глобальные переменные – это те переменные, которые объявляются вне функций. Глобальные переменные можно использовать в любой директиве, независимо от того, в какой функции или в каком блоке эта директива используется.

Параметры, которые перечисляются в заголовке функции, относятся к локальным переменным. **Область видимости локальных переменных** – только сама функция, в других функциях эти переменные не доступны. По завершении функции локальные переменные, в том числе и объявленные в строке параметров, удаляются из памяти.

Имена локальных переменных могут совпадать с именами локальных переменных других функций и даже с именами глобальных переменных. В последнем случае локальные переменные перекрывают глобальные (т.е. глобальная переменная, имя которой совпало с именем локальной в данной функции, становится недоступна).

Функция завершается, когда будут выполнены все ее инструкции или когда встретится оператор **return**. Если функция должна возвращать какое-либо значение, то в операторе **return** должно присутствовать выражение (или переменная, или константа):

`return a;` – возвращается значение переменной.

`return 0;` – возвращается значение 0.

`return (a + b)/2;` – возвращается значение, вычисленное по формуле.

В теле функции может быть несколько операторов **return** или не быть вовсе (если функция ничего не возвращает).

Возвращаемое функцией значение может быть присвоено какой-либо переменной вызывающей функции:

```
key = getch();
```

или проигнорировано. Например, если код нажатой клавиши при вызове этой функции не нужен, можно написать просто:

```
getch();
```

Разработка функции

Функции обычно создаются на этапе проектирования, когда вся программа разделяется на задачи и подзадачи, которые и оформляются в виде отдельных функций.

При разработке функции, прежде всего, нужно:

– определить ту задачу, которую должна выполнять функция;
– определить, какие аргументы нужны ей для решения этой задачи;
– определить, должна ли функция возвращать какую-либо информацию.

Далее необходимо **создать заголовок функции**, для этого:

– придумать имя функции;
– решить вопрос, какие переменные необходимо передавать в качестве параметров в будущую функцию, выбрать их типы и задать имена;
– решить вопрос о типе возвращаемого значения функции.

Следующий шаг – **создание тела функции**, при этом нужно:

– разработать алгоритм выполнения задачи, решаемой функцией;
– определить, какие переменные будут являться локальными в функции, и объявить их;
– затем разработать тело функции в соответствии с решаемой ею задачей, с учетом параметров функции и возвращаемого значения.

В заголовочном файле или в верхней части основного файла разместить предварительное объявление функции. Теперь функцию можно использовать в своей программе.

Пример 1. Разработать функцию возведения вещественного числа в целую степень: Функция должна использоваться, например, для таких вычислений:

$y = 3.1415^4$.

Для работы функции понадобятся два аргумента: само число, которое возводится в степень, (вещественное, выберем тип `float`) и показатель степени (целое, положительное, выберем тип `unsigned short`). Возвращать функция должна вещественное число, значит тип возвращаемого значения – тоже вещественное (`float`). Выберем имя функции – `Power`.

Заголовок будет выглядеть так:

```
float Power(float Value, unsigned short p)
```

Разрабатываем тело функции. Используем для возведения в степень обычное перемножение в цикле нужное число раз. Для этого нам понадобятся две локальные переменные – счетчик цикла и переменная, в которой будет формироваться вычисляемое значение.

```
float Power(float Value, unsigned short p)
{
  unsigned short i;
  float f = 1;
  for (i = 0; i < p; i++)
    f *= Value;
  return f;
}
```

В заголовочном файле или в верхней части основного файла размещаем прототип функции:

```
float Power(float Value, unsigned short p);
```

Использовать функцию можно будет так:

```
float y;
y = Power(3.1415, 4);  // y = 3.1415^4
```

Еще несколько примеров:

Функция определения длины строки.

```
int MStrLen(const char* s)
{
  int i = 0;
  while (s[i] != 0)        //  или просто while (s[i])
    i++;
  return i;
}
```

Использование функции:

```
int k = MStrLen("qwerty"); // Возвращает число 6
```

Функция определения количества цифр в целом числе.

```
int DigCount(int Value) // Кол-во цифр в числе
{
  int i;
  for(i=0; Value!=0; i++)     // Или так: for(i=0; Value; i++)
    Value /= 10;
  return i;
}
```

Функция выделения n-ой справа цифры в целом числе.

```
int DigValR(int Value, int n)
{
  int i;
  for(i=1; Value != 0; i++)
  {
    if (i == n)
      return Value % 10;
    Value /= 10;
  }
  return -1;
}
```

Функция выделения n-ой слева цифры в целом числе.

```
int DigValL(int Value, int n)
{
  int i;
  int count = DigCount(Value);
  for(i=0; Value != 0; i++)
  {
    if (i == count - n)
      return Value % 10;
    Value /= 10;
  }
  return -1;
}
```

Использование функций:

```
int a = DigCount(125);   // 3
int b = DigValR(125, 3); // 1
int c = DigValL(125, 3); // 5
```

Типичные ошибки при определении и вызове функций:

```
//Основная программа
float y, f;
f = 25;
y = Square(f);
```

```
void Square(float f)
{
  f = f * f;
}
```

Функция должна возвращать вычисленное значение, а тип возвращаемого значения, указанный в заголовке – `void`.

```
//Основная программа
float y, f;
f = 25;
y = Square(f);
```

```
float Square(float f)
{
  f = f * f;
}
```

Функция должна возвращать вычисленное значение, а оператора **`return`** нет. Правильный вариант функции:

```
float Square(float f)
{
  f = f * f;
return f;
}
```

или даже:

```
float Square(float f)
{
  return f * f;
}
```

При **разработке функции** можно поступить и по-другому: оформить в виде функции некоторый готовый фрагмент программы. Для этого необходимо:

Скопировать этот фрагмент и выделить как блок операторов (с помощью **{}**). Это будет заготовка для содержимого будущей функции.

Создать заголовок функции, для чего придумать имя, решить вопрос, какие переменные необходимо передавать в качестве параметров в будущую функцию и включить их в заголовок, рушить вопрос о возвращаемом значении функции..

Определить, какие переменные будут являться локальными в функции, и объявить их.

Доработать тело функции в соответствии с требованиями, с учетом новых имен локальных переменных, параметров функции и возвращаемого значения.

В заголовочном файле или в верхней части основного файла разместить предварительное объявление функции.

В вызывающей программе фрагмент кодов, оформленный в виде функции, заменить обращением к этой функции.

Пример 2. В лабораторной работе № 9 рассматривался пример программы для заполнения массива случайными числами, в которой массив заданного размера сначала заполнялся случайными числами, а затем выводился на дисплей.

```
#define NMAX 100
#define NMIN -100
#define SIZE 20

int main()
{
  int Array[SIZE];
  int i;
```

```
  srand(time(0));
  for (i = 0; i < SIZE; i++)// Заполнение масс. случ. числами
    Array[i] = (float)rand()/RAND_MAX * (NMAX - NMIN) + NMIN;
  for (i = 0; i < SIZE; i++)
    printf ("%d\n", Array[i]); // Вывод на дисплей
  getch();
}
```

В программе можно выделить две основные части: заполнение массива случайными числами и вывод массива на дисплей. Их можно оформить в виде отдельных функций, это позволит не только структурировать данную программу, но и использовать данные функции в других программах, где понадобится выполнять подобные действия. Объединять же генерацию и вывод в одну функцию – плохая идея, т.к. это ограничит применяемость функции.

Выделяем блок операторов для первой функции:

```
{
  for (i = 0; i < SIZE; i++)// Заполнение масс. случ. числами
    Array[i] = (float)rand()/RAND_MAX * (NMAX - NMIN) + NMIN;
}
```

Начальную инициализацию генератора псевдослучайной последовательности **srand(time(0))** в функцию включать нецелесообразно, т.к. делать инициализацию в программе нужно только один раз, а вызывать функцию заполнения массива случайными числами можно несколько раз.

Создаем заголовок: придумываем название функции, например: **FillArrayRand**. Определяем аргументы функции: во-первых, сам массив, который нужно заполнить, во-вторых, его длина. Возвращаемого значения у данной функции нет, поэтому пишем **void**.

```
void FillArrayRand(int Arr[], unsigned int Len)
```

Локальная переменная **i** – счетчик в цикле. Окончательно получаем:

```
void FillArrayRand(int Arr[], unsigned int Len)
{                       // Заполнение масс. случ. числами
  for (int i = 0; i < Len; i++)
    Arr[i] = (float)rand()/RAND_MAX * (NMAX - NMIN) + NMIN;
}
```

Эта функция заполняет массив, передаваемый в функцию в качестве аргумента, случайными числами в диапазоне, который задается константами **NMIN** и **NMAX**. В реальной функции желательно диапазон тоже передавить в функцию в качестве параметров, например так:

```
void FillArrayRand(int Arr[], unsigned int Len, int Min, int Max)
```

В этом случае функция будет более гибкой.

Выделяем блок операторов для второй функции:

```
{
  for (i = 0; i < SIZE; i++)
    printf ("%d\n", Array[i]); // Вывод на дисплей
}
```

Заголовок: придумываем название – **ViewArray**, аргументы – те же что и для предыдущей функции. Окончательно получаем:

```
void ViewArray (int Arr[], unsigned int Len)
{
  for (int i = 0; i < Len; i++)
    printf ("%d\n", Arr[i]); // Вывод на дисплей
}
```

С использованием разработанных функций программа будет выглядеть более структурировано:

```
#define NMAX 100
#define NMIN -100
#define SIZE 20

void FillArrayRand(int Arr[], unsigned int Len); // Прототипы
void ViewArray(int Arr[], unsigned int Len);     // функций

int main()
{
  int Array[SIZE];

  srand(time(0));
  FillArrayRand(Array, SIZE); // Вызов ф-ии заполнения массива
  ViewArray(Array, SIZE);  // Вызов функции вывода массива
  getch();
}
//--------------------------------------
void FillArrayRand(int Arr[], unsigned int Len)
{
  for (int i = 0; i < Len; i++)// Заполнение масс. случ. числами
    Arr[i] = (float)rand()/RAND_MAX * (NMAX - NMIN) + NMIN;
}
//--------------------------------------
void ViewArray (int Arr[], unsigned int Len)
{
  for (int i = 0; i < Len; i++)
    printf ("%4d\n", Arr[i]); // Вывод на дисплей
}
//--------------------------------------
```

Пример 3. В лабораторной работе № 9 рассматривался следующий пример: «Дан одномерный массив размерностью **N**. Подсчитать количество отрицательных элементов в массиве, найти **k**–й отрицательный элемент массива». Разработанная программа выглядела так:

```
#define N 20  // размер массива

int main(int argc, char* argv[])
{
  int i, Arr[N];
  int kn = 0;
  int ki = -1, kneg;

  setlocale(LC_ALL, "Russian");
  printf("Введите элементы массива : \n");
  for (i = 0; i < N; i++) // Ввод массива с клавиатуры
  {
    printf("Arr[%d] = ", i); // Подсказка для пользователя
    scanf("%d", &Arr[i]); // Ввод элемента массива
  }
  printf("Введите номер для поиска отрицательного элемента: ");
  scanf("%d", &kneg);

  for (i = 0; i < N; i++)  // Основной цикл
  {
```

```
    if (Arr[i] < 0) // Если очередной элемент отрицательный
    {
      kn++;           // Число отриц. элементов в массиве
      if (kn == kneg) // Если заданный отриц. элемент массива
        ki = i;       // Запоминаем его индекс в  массиве
    }
  }

  if (kn != 0)
    printf("Число отриц.элементов в массиве = %d\n", kn);
  else
    printf("Отрицательных элементов в массиве нет!\n");
  if (ki >= 0)
    printf("Arr[%d] = %d является %d-ым отриц.элем.массива\n",
      ki, Arr[ki], kneg);
  else
    printf("Отриц.элем.с порядковым номером %d нет\n", kneg);

  _getch();
  return 0;
}
```

В задании были дополнительные требования: добавить в программу выбор способа заполнения массива (вручную, случайными числами или по возрастанию, начиная от **x1**, с шагом **k**, где **x1** и **k** задаются с клавиатуры) и цикл **do-while** для организации повторных вычислений.

Перепишем программу с использованием функций. Для этого проанализируем текст программы. Сначала массив должен заполняться данными, желательно тремя разными способами по выбору пользователя, следовательно, лучше всего для каждого из этих способов создать отдельную функцию.

Одна – заполнение случайными числами – уже нами написана, и мы можем ее использовать и в этой программе. Для создания второй – заполнение вручную – выделим соответствующие строки программы:

```
  printf("Введите элементы массива : \n");
  for (i = 0; i < N; i++) // Ввод массива с клавиатуры
  {
    printf("Arr[%d] = ", i); // Подсказка для пользователя
    scanf("%d", &Arr[i]); // Ввод элемента массива
  }
```

и оформим их в виде функции. Для этого создаем заголовок, выбираем имя функции – **FillArrayHand**, аргументы будут такими же, как у функции **FillArrayRand**.

```
  void FillArrayHand(int Arr[], unsigned int Len)
  {
    printf("Введите элементы массива: \n");
    for (int i = 0; i < Len; i++) // Ввод массива с клавиатуры
    {
      printf("Arr[%d]= ", i); // Подсказка для пользователя
      scanf("%d", &Arr[i]); // Ввод элемента массива
    }
  }
```

Аналогично можно создать и третью функцию – заполнение массива значениями по возрастанию, начиная от **x1**, с шагом **k**.

После заполнения массива его нужно вывести на дисплей, для этого используем уже разработанную функцию **ViewArray**.

Следующий шаг – обработка массива в соответствии с заданием: подсчитать количество отрицательных элементов в массиве, найти **k**–й отрицательный элемент массива. Эту задачу лучше всего разделить на две подзадачи и оформить две функции: сначала в цикле подсчитать количество отрицательных элементов, а затем – во второй функции – искать **k**–й отрицательный элемент.

Первая функция должна просмотреть весь массив, подсчитать количество отрицательных элементов в массиве и вернуть это число. Аргументы будут такими же, как у функции `FillArrayRand`, а тип возвращаемого значения будет `int`.

```
int CountNegativ(int Arr[], unsigned int Len)
{
  int i, kn;
  kn = 0;
  for (i = 0; i < Len; i++)
  {
    if (Arr[i] < 0)
      kn++; // Число отриц. элементов в массиве
  }
  return kn;
}
```

Вторая функция тоже просматривает весь массив и ищет **k**–й отрицательный элемент, индекс которого должна вернуть в качестве возвращаемого значения. Если таковой не найден – возвращаем –1. Отрицательное значение позволит показать, что нужный элемент не найден. Номер искомого элемента тоже нужно передать в функцию в качестве аргумента, поэтому список параметров функции будет иным:

```
int FindNegativ(int Arr[], unsigned int Len, unsigned int k)
{          // k -  номер отрицательного элемента, который ищем
  int i, kn;
  kn = 0;
  for (i = 0; i < Len; i++)
  {
    if (Arr[i] < 0)
    {
      kn++; // Число отриц. элементов в массиве
      if (kn == k) // Если заданный отриц. элемент массива
        return i;   // Возвращаем его индекс в массиве
    }
  }
  return -1; // Если не нашли
}
```

Теперь можно, используя разработанные функции, написать текст самой программы:

```
#include <stdio.h>
#include <locale.h>
#include <conio.h>
#include <cstdlib>
#include <time.h>
                          // Прототипы функций
void FillArrayRand(int Arr[], unsigned int Len);
void FillArrayHand(int Arr[], unsigned int Len);
void ViewArray(int Arr[], unsigned int Len);
int CountNegativ(int Arr[], unsigned int Len);
int FindNegativ(int Arr[], unsigned int Len, unsigned int k);
```

```
#define NMAX 100
#define NMIN -100
#define N 20  // размер массива

int main(int argc, char* argv[])
{
  int Array[N], kneg, ineg;
  char key;

  srand(time(0));
  setlocale(0, "Russian");
  for (;;) // Бесконечный цикл
  {
    printf("\n\nВыберите способ заполнения массива:\n");
    printf("  0 - выход из программы\n");
    printf("  1 - вручную\n");
    printf("  2 - случайными числами\n");
    printf("  3 - по возрастанию\n");
    key = _getch();
    switch (key)
    {
    case '0':
      return 0;   // Завершение программы
    case '1':
      FillArrayHand(Array, N); // Заполнение массива вручную
      break;
    case '2':
      FillArrayRand(Array, N);// Заполнение массива случ.числами
      break;
    case '3':
      //FillArrayInc(Array, N); // Эту функцию еще надо написать
      break;
    default:
      FillArrayRand(Array, N);
      break;
    }
    ViewArray(Array, N);  // Вывод массива на дисплей

    kneg = CountNegativ(Array, N); // Считаем отриц. элементы
    if (kneg == 0)
      printf("Отрицательных элементов в массиве нет!\n");
    else
    {
      printf("Число отриц. элементов в массиве = %d\n", kneg);
      printf("Введите номер для поиска отрицательного элемента: ");
      scanf_s("%d", &kneg);
      ineg = FindNegativ(Array, N, kneg); // Ищем нужный элем.
      if (ineg >= 0)
        printf("Array[%d] = %d является %d-ым отриц.
                элем.массива\n", ineg, Array[ineg], kneg);
      else
          printf("Отриц. элементов с номером %d нет\n", kneg);
    }
  } // Конец цикла for
}
```

ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

в начале программы ОБЯЗАТЕЛЬНО выводить:
ФИО, группа, номер лаб. работы, номер варианта.

Задание 1. Разработать функцию из следующего списка и программу, демонстрирующую ее работу. В программе обязательно должен быть цикл для организации повторных вычислений. Вариант задания выдает преподаватель.

Варианты заданий

1. Разработать функцию, вычисляющую сумму цифр целого числа.

2. Разработать функцию, вычисляющую количество нечетных цифр целого числа.

3. Разработать функцию, возвращающую целое число, цифры в котором переставлены, например, было число 12345, возвращает число 54321.

4. Разработать функцию, вычисляющую среднее цифр целого числа.

5. Разработать функцию, вычисляющую среднее арифметическое чисел от a до b включительно.

6. Разработать функцию, вычисляющую среднее геометрическое чисел от a до b включительно.

7. Разработать функцию, вычисляющую квадратный корень вещественного числа S по алгоритму $x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$ с заданной точностью eps.

8. Разработать функцию, возвращающую дробную часть вещественного числа, округленного до k-того знака после запятой.

9. Разработать функцию, вычисляющую расстояние от начала координат до точки с координатами x, y, z.

10. Разработать функцию, вычисляющую расстояние между двумя точками на плоскости.

11. Разработать функцию, возвращающую максимальную цифру целого числа.

12. Разработать функцию, возвращающую минимальную цифру целого числа.

13. Разработать функцию, вычисляющую произведение цифр целого числа.

14. Разработать функцию, определяющую, есть ли среди цифр числа одинаковые.

15. Разработать функцию, определяющую, делится ли сумма цифр числа на 3.

16. Разработать функцию, определяющую, делится ли произведение цифр числа на 7.

17. Разработать функцию, возвращающую четверть, в которой находится точка с координатами x,y.

18. Разработать функцию, определяющую, является ли целое число палиндромом, т. е. таким числом, десятичная запись которого читается одинаково справа налево и слева направо.

19. Разработать функцию, вычисляющую количество четных цифр целого числа.

20. Разработать функцию, определяющую, является ли целое число простым.

Задание 2. Переделать программу из лабораторной работы № 9 с использованием функций и меню. Добавить в программу выбор способа заполнения массива: вручную или случайными числами. В программе обязательно должен быть цикл для организации повторных вычислений.

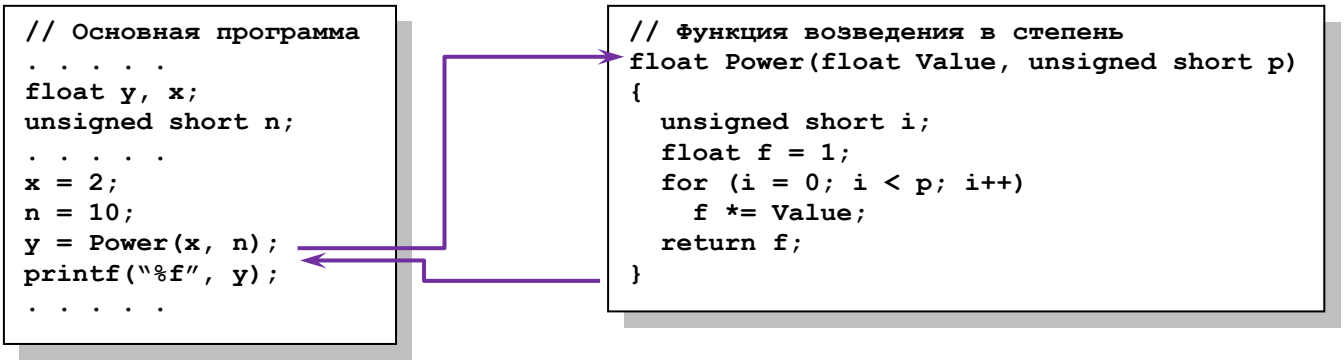
Для получения максимального балла необходимо добавить в программу дополнительно функции заполнения массива:

*– по возрастанию, начиная от **x1,** с шагом **k**, где **x1** и **k** задаются с клавиатуры*

*– случайными числами в диапазоне от **min** до **max**, где **min** и **max** задаются с клавиатуры.*

Вызов функции и передача параметров через стек

Передача параметров в функцию и возвращаемого значения из функции – процесс, осуществляемый через стек. Чтобы понять, как это происходит рассмотрим пример вызова функции:

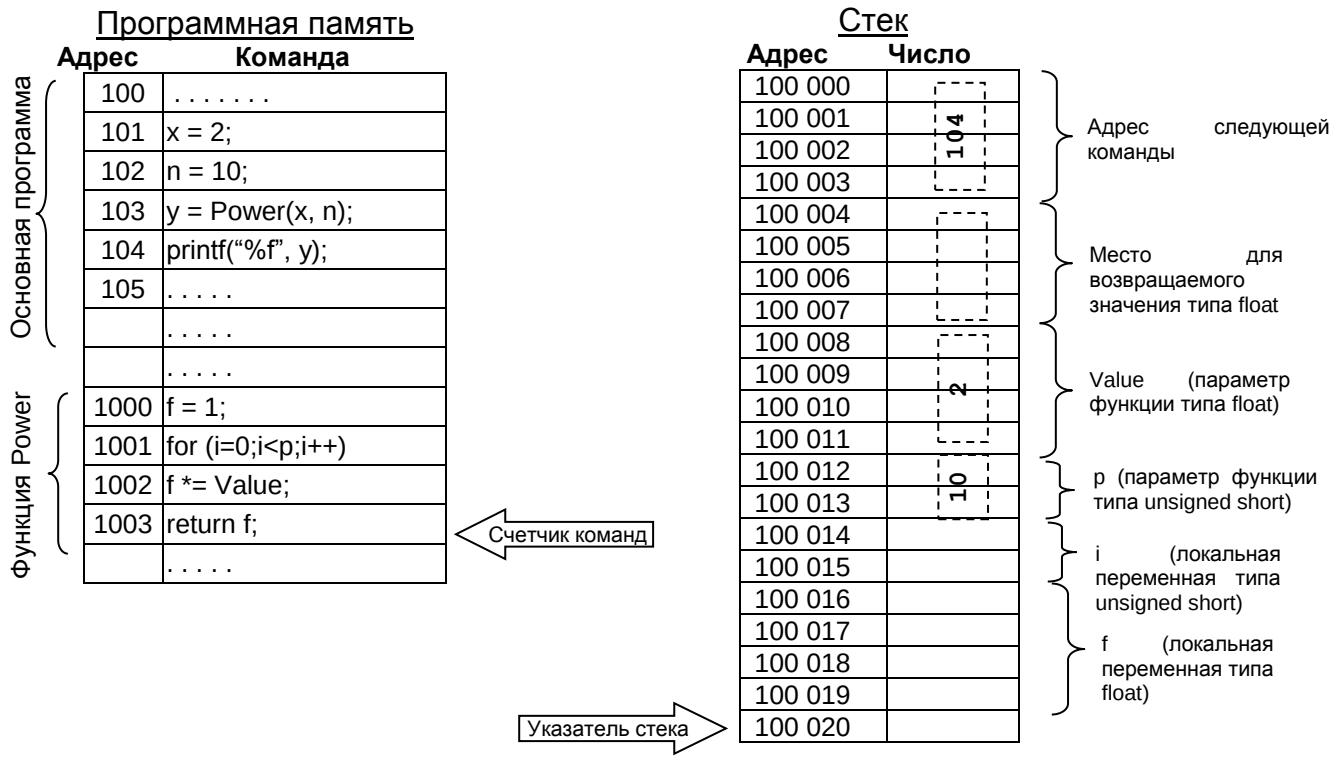


Программная память хранит коды команд (конечно же в откомпилированном виде) и каждая команда имеет свой адрес. В специальном регистре процессора, так называемом **счетчике команд**, хранится адрес следующей команды, которую нужно исполнить процессору.

Под переменные отводится специальная память – стек. В ней тоже каждый байт имеет свой адрес, кроме того имеется специальный регистр – указатель стека. Стек организован по принципу: первый пришел – последний вышел, т.е. ячейки, записанные первыми удаляются в последнюю очередь. При записи очередного байта (или нескольких байт, в соответствии с размером переменной) указатель стека перемещается на соответствующее число байт.

Вызов функции (например, **`Power`**):

1. В стек записывается адрес следующей команды (напр., 104). Указатель стека перемещается вниз на 4 байта.
2. В стеке резервируется место под возвращаемое значение (ячейки 100004-100007) . Указатель стека перемещается вниз на 4 байта.
3. В стеке выделяется место под параметры функции и туда копируются значения аргументов (4 байта под **`Value`** и 2 байта под **`p`** и записываются значение **`x`** равное 2 и **`n`** равное 10). Указатель стека перемещается вниз на 6 байт.
4. В счетчик команд записывается адрес первой команды функции (напр., 1000).
5. В стеке выделяется место под локальные переменные функции (2 байта под `i` и 4 байта под **`f`**). Указатель стека перемещается вниз на 6 байт.
6. Начинается выполнение функции (пока не дойдет до конца или до оператора **`return`**).
7. В ячейки стека, зарезервированные под возвращаемое значение, копируется вычисленное значение (переменной, указанной в операторе **`return`**, – число 1024).
8. Освобождаются ячейки, выделенные для локальных переменных и параметров функции. Указатель стека перемещается вверх на 12 байт.
9. В счетчик команд записывается адрес, ранее записанный в стек (104).
10. Возвращаемое значение из стека присваивается переменной (**`y`**).
11. Указатель стека перемещается вверх на 8 байт.
12. Продолжается выполнение команд с адреса, на который указывает счетчик команд.



Чем больше параметров и сложнее их тип, чем сложнее тип возвращаемого значения, тем больше расходуется памяти. Для сокращения затрат при передаче сложных параметров используют указатели или ссылки.