Программирование на языке Си

Лекция 7.
Функции

ЗАЧЕМ НУЖНЫ ФУНКЦИИ?

Самая важная причина использования функций заключается в том, что функции существенно упрощают разработку, отладку и поддержку программ. Функции позволяют избежать дублирования кода, запутанных логических конструкций и глубоко вложенных циклов. В конечном счете, использование функций делает программу легко читаемой и понятной даже программисту, который не участвовал в ее разработке.

ЧТО ТАКОЕ ФУНКЦИЯ?

Функция – это часть программы (или подпрограмма), которая решает одну конкретную задачу. Точнее, данное определение относится к *грамотно* написанной функции, но именно к проектированию таких функций мы стремимся. Функция может производить какие-то вычисления и возвращать полученный результат, а может просто выполнять последовательность действий и ничего не возвращать. Функция, которая не возвращает значения, в некоторых языках программирования называется *процедурой*. В объектно-ориентированном программировании функции называются *методами*.

ОПРЕДЕЛЕНИЕ ФУНКЦИИ

Рассмотрим, как определить функцию на примере вычисления среднего арифметического трех чисел:

```
float getAverage(float x, float y, float z) {
  float sum = x+y+z;
  return sum/3;
}
int main() {
  printf("%f\n", average(12.5,56.7,66.8));
}
```

Определение функции состоит из нескольких частей:

- тип возвращаемого значения;
- название функции;
- формальные параметры;
- тело функции;
- оператор `return`.

ТИП ВОЗВРАЩАЕМОГО ЗНАЧЕНИЯ

Тип возвращаемого значения может быть любым типом, который используется при объявлении переменных; в нашем примере – это тип `float`:

```
float getAverage(float x, float y, float z) {
  …
}
```

Указание возвращаемого типа сообщает программисту, как нужно вызывать функцию:

```
printf("%f\n", getAverage(12.5,56.7,66.8));
```

В этом примере мы используем спецификатор `%f`, поскольку в качестве возвращаемого значения указан тип `float`. Можно также использовать запись

```
float avg = average(12.5,56.7,66.8);
```

НАЗВАНИЕ ФУНКЦИИ

Название функции – это идентификатор, поэтому оно должно подчиняться определенным правилам: состоять из букв, цифр или знака подчеркивания и начинаться с буквы или со знака подчеркивания. Название функции может быть любым в рамках данных правил, но оно должно точно указывать на ту задачу, которую решает функция. Например, если функция предназначена для сортировки массива, то подходящим названием может быть `sort_array` или `sort`, а имена типа `fg`, `f1` или `mas` никак не связаны данной задачей, поэтому их использование нежелательно.

Название функции указывается при ее определении:

```
float getAverage(float x, float y, float z) {
  …
}
```

и при вызове:

```
printf("%f\n", getAverage(12.5, 56.7, 66.8));
```

ФОРМАЛЬНЫЕ АРГУМЕНТЫ

После указания имени функции в круглых скобках перечисляются *формальные* аргументы этой функции:

```
float getAverage(float x, float y, float z) {
   …
}
```

Формальные аргументы указывают на то, значения каких типов можно передать в функцию при вызове:

```
printf("%f\n", getAverage(12.5, 56.7, 66.8));
```

ФАКТИЧЕСКИЕ АРГУМЕНТЫ

Аргументы функции, указываемые при вызове, называются *фактическими*; их количество, порядок и тип должны быть такими же, как и у формальных аргументов; если данное требование не выполняется, то компилятор выдаст ошибку. Например, следующие строки кода приведут к ошибкам:

```
printf("%f\n", getAverage(12.5));
printf("%f\n", getAverage("1.5", 6.7, 6.8));
printf("%f\n", getAverage());
```

ТЕЛО ФУНКЦИИ

После указания формальных аргументов в фигурных скобках находится *тело функции*; здесь идут все операторы, которые должна выполнить функция:

```
float getAverage(float x, float y, float z) {

  float summa = x+y+z;
  return summa/3;
}
```

ОПЕРАТОР RETURN

Для остановки работы функции и возвращения ее значения предназначен оператор `return`. В зависимости от функции оператор `return` может присутствовать один или несколько раз, а может и вовсе отсутствовать. Стандартный способ использования оператора `return` – это указание его в конце функции с целью возвращения ее значения; именно так сделано в нашей функции `getAverage`:

```
float getAverage(float x, float y, float z) {
  float summa = x+y+z;
  return summa/3;
}
```

НЕСКОЛЬКО ОПЕРАТОРОВ RETURN

Два раза оператор `return` используется в следующей функции, которая определяет максимум из двух чисел:

```
int max(int m, int n) {
  if (m>n) {
    return m;
  } else {
    return n;
  }
}
```

ОТСУТСТВИЕ RETURN

Если функция не возвращает никакого значения, другими словами, при ее определении возвращаемым типом указан `void`, то оператор `return` может отсутствовать. Примером такой функции является функция вывода массива чисел на экран:

```
void printNumbers(int numbers[], int size) {
  for (int i=0; i<size; i++) {
    printf("%d ", numbers[i]);

   }
}
```

ОШИБКА!

Если же при определении функция тип возвращаемого значения указан отличным от `void`, то хотя бы один оператор `return` должен присутствовать; более, все ветки логических конструкций должны содержать оператор `return`. Например, функция

```
int max(int m, int n) {
  if (m>n) {
    return m;
  } else {
    printf("%d\n", n);
  }
}
```

определена неправильно, поскольку ветка `else` не возвращает никакого значения, в то время как функция должна возвращать значение типа `int`.

ГЛОБАЛЬНЫЕ И ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

```
int x = 10;
int summa(int y) {
  result = x+y;
  return x + y;
}
```

А теперь еще один:

```
int summa_version2(int y) {
  int x = 10;
  result = x+y;
  return result;
}
```

ГЛОБАЛЬНЫЕ И ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

Функция, как и любая другая конструкция языка Си, делит переменные на глобальные и локальные. *Локальными* являются те переменные, которые описаны внутри функции и, значит, доступны только внутри этой функции. Например, в следующем фрагменте кода переменная result – локальная:

```
int summa(int a, int b) {
    int result = a+b;
    return result;
}
```

ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

Понятие локальной переменной относится не только к функциям, но и к другим конструкциям языка. Например, в следующем фрагменте переменная **i** является локальной для цикла:

```
for (int i=0; i<10; i++) {
  …
}
```

Переменная, описанная внутри некоторого блока, ограниченного фигурными скобками, является локальной для этого блока:

```
if (…) {
  …
  int x;
  …
}
```

Какие переменные использовать?

Используйте глобальные переменные только при для хранения глобальных данных.

Например,

- Название фирмы.
- Константа Пи.
- …

РЕКУРСИВНЫЕ ФУНКЦИИ

Существует важный класс функций, называемых рекурсивными. Функция называется *рекурсивной*, если она вызывает сама себя. Хотя любую рекурсивную программу можно переписать без использования рекурсии, в ряде случаев рекурсивная версия функции имеет более простой вид, особенно в тех случаях, когда вычисляемое значение или объект, используемый в функции, определяется рекурсивно.

Рассмотрим функцию, вычисляющую факториал. *Факториал* числа ***N*** обозначается через ***N!*** и равен произведению ***1*2*3*4*…*N***; по определению также полагается ***0!=1***. Очевидная реализация такой функции следующая:

НЕРЕКУРСИВНАЯ РЕАЛИЗАЦИЯ ФАКТОРИАЛА

```
int factorial(int n) {
  int result = 1;
  for (int i=1; i<=n; i++) {
    result *= i;
  }
  return result;
}
```

РЕКУРСИВНАЯ РЕАЛИЗАЦИЯ ФАКТОРИАЛА

Заметим теперь, что факториал можно задать двумя формулами: ***0!=1*** и ***N!=N*(N-1)!*** В данном случае значение задается только для минимального ***N***, т.е. для ***0***, а для остальных значения функция определяется через предыдущее значение ***N***. Например, ***6!=6*5!*** Используя данные соотношения функцию вычисления факториала можно модифицировать:

```
int factorial(int n) {
  if (n==0) {
    return 1;
  } else {
    return n*factorial(n-1);
  }
}
```

БОЛЕЕ КОРОТКИЙ ВАРИАНТ

Можно также воспользоваться оператором ?: и сократить размер функции:

```
int factorial(int n) {
  return n==0 ? 1 : n*factorial(n-1);
}
```

Обратите внимание на то, что такая реализация не использует цикл, зато она вызывает сама себя: `n*factorial(n-1)`.

ПРОТОПИПЫ ФУНКЦИЙ

Описание функции состоит из двух частей: объявление и реализация. Прототип функции сообщает компилятору, что такая функция есть, и ее реализация находится где-то ниже.

```
float getAverage(float x, float y, float z);

int main() {
    printf("%f\n", getAverage(1.2,3.4,5.6));
}

float getAverage(float x, float y, float z) {
    return (x+y+z)/3;
}
```

СОЗДАНИЕ ЗАГОЛОВОЧНОГО ФАЙЛА

Мы уже сталкивались с заголовочными файлами, когда подключали библиотеку `stdio.h` для работы с функциями ввода/вывода или библиотеку `stdlib.h` для работы с функцией `rand()` для генерации псевдослучайных чисел. Эти функции были кем-то реализованы, и любой программист может их использовать. Другими словами, функция написана один раз, а использована она может быть использована много раз и в разных программах.

По аналогии с этими библиотеками можно создать и свою библиотеку, куда поместить часто используемые функции. Для этого нужно выполнить следующие действия:

- создать файл с расширением .h;
- поместить в него необходимые функции;
- подключить файл в программе;
- вызвать требующуюся функцию.

МАССИВЫ В КАЧЕСТВЕ АРГУМЕНТОВ ФУНКЦИЙ

Массивы могут быть аргументами функций. Чтобы передать массив в качестве аргумента, нужно указать имя и пустые скобки, например, следующая функция вычисляет сумму элементов массива:

```
float getSum(float numbers[], int size) {
  float result = 0;
  for (int i=0; i<size; i++) {
    result += numbers[i];
  }
  return result;
}

int main() {
  const int N = 5;
  float weights[N] = {
    3.5, 6.1, 6.3, 9.3, 7.7
  };
  printf("%.1f\n", getSum(weights, N));
  return 0;
}
```

МАТРИЦЫ КАК АРГУМЕНТЫ

При передаче матриц требуется указывать второй размер, первый размер можно указывать, а можно не указывать. Следующая функция проверяет, является ли квадратная матрица диагональной. Матрица называется диагональной, если все ее элементы за исключением диагональных равны нулю. Диагональные элементы определяются тем, что оба их индекса равны. Функция имеет следующий вид:

```
bool isDiagonal(float matrix[N][N]) {
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      if (i!=j && matrix[i][j] != 0) {
        return false;
      }
    }
  }
  return true;
}
```

ПРИЕМЫ СОКРАЩЕНИЯ КОДА

```
bool isEven(int n) {
  if (n%2==0) {
    return true;
  } else {
    return false;
  }
}
```

Обратите внимание на то, что, если убрать ветку `else`, то работа функции не изменится, поскольку оператор `return` останавливает работу функции:

```
bool isEven(int n) {
  if (n%2==0) {
    return true;
  }
  return false;
}
```

ЕЩЕ КОРОЧЕ

Еще сократить код можно при помощи оператора ?::

```
bool isEven(int n) {
    return n%2==0 ? true : false;
}
```

Однако оптимальный вариант выглядит так:

```
bool isEven(int n) {
  return n%2 == 0;
}
```

ЕЩЕ ОДИН ПРИМЕР

Рассмотрим еще один пример лаконичной записи функции, определяющей максимум из двух чисел. Очевидный способ следующий:

```
int max(int m, int n) {
  if (m>n) {
    return m;
  } else {
    return n;
  }
}
```

Сократить код можно так:

```
int max(int m,int n) {
   return m>n ? m : n;
}
```

ПРИМЕР С ЦИКЛОМ

Еще один пример, использующий то, что оператор `return` прерывает работу функции, – это поиск элемента в массиве. Он может быть реализован так:

```
int indexOf(int element) {
  for (int i=0; i<n; i++) {
    if (numbers[i]==element) {
      return i;
    }
  }
  return -1;
}
```

Возвращение функцией индекса ***-1*** является общепринятым способом сообщить, что элемент не найден.

ПРАВИЛЬНОЕ ПРОЕКТИРОВАНИЕ ФУНКЦИЙ

Главное требование, которое нужно соблюдать при разработке функций, заключается в следующем. *Функция должна решать ровно одну определенную задачу, и название функции должно точно отражать суть этой задачи.* По-настоящему оценить значимость данного требования можно только при наличии опыта разработки больших приложений, однако заметить некоторые преимущества соблюдения этого принципа можно даже на небольших примерах.

Рассмотрим некоторую программу, которая предназначена для работы с массивами; в такой программе могут присутствовать инициализация массива случайными элементами, вывод массива на экран, сортировка массива и еще один вывод:

ПРИМЕР «ПОРТЯНКИ»

```
for (int i=0; i<size; i++) {
  numbers[i] = rand()%100;
}

for (int i=0; i<size; i++) {
  printf("%d ", numbers[i]);
}

for (int i=0; i<size-1; i++) {
  for (int j=size-1; j>0; j--) {
    if (numbers[j] < numbers[j-1]) {
      int temp = numbers[j];
      numbers[j] = numbers[j-1];
      numbers[j-1] = temp;
    }
  }
}
for (int i=0; i<size; i++) {
  printf("%d ", numbers[i]);
}
```

ПРИМЕР ХОРОШЕЙ ОРГАНИЗАЦИИ КОДА

```
void initNumbers(int numbers[], int size);
void printNumbers(int numbers[], int size);
void sortNumbers(int numbers[], int size);

int main() {
  const int mySize = 10;
  int myNumbers[mySize];
  initNumbers(myNumbers, mySize);
  printNumbers(myNumbers, mySize);
  sortNumbers(myNumbers, mySize);
  printNumbers(myNumbers, mySize);
}
```