Лабораторная работа №7

Функции. Строки.

Цель работы: освоить принципы работы с функциями и строками в Си, выполнить упражнения по вариантам

Краткие теоретические сведения

Большинство компьютерных программ, решающих реальные практические задачи, намного превышают те программы, которые были представлены в предыдущих примерах. Экспериментально показано, что наилучшим способом создания и поддержки больших программ является их конструирование из маленьких фрагментов, или *модулей,* каждый из которых более управляем, чем сложная программа.

Модули в С называются *функциями.* Обычно программы на С пишутся путем объединения новых функций, которые пишет сам программист, с функциями, уже имеющимися в стандартной библиотеке С.

Стандартная библиотека С обеспечивает широкий набор функций для выполнения типовых математических расчетов, операций со строками, с символами, ввода-вывода, проверки ошибок и многих других полезных операций. Это облегчает работу программиста, поскольку эти функции обладают многими из необходимых программисту свойств. Функции стандартной библиотеки С являются частью среды программирования С++.

Программист может написать функции, чтобы определить какие-то специфические задачи, которые можно использовать в разных местах программы. Эти функции иногда называют *функциями, определенными пользователем.* Операторы, реализующие данную функцию, пишутся только один раз и скрыты от других функций.

Функция *активизируется* (т. е. начинает выполнять запроектированную для нее задачу) путем *вызова функции.* В вызове функции указывается ее имя и дается информация (в виде *аргументов),* необходимая вызываемой функции для ее работы. Это аналогично иерархической форме управления. Начальник *(вызывающая функция* или *вызывающий оператор)* просит подчиненного *(вызываемую функцию)* выполнить задание и *возвратить* (т.е. сообщить) результаты после того, как задание выполнено. Функция-начальник не знает, *как* функция-подчиненный выполняет порученное ей задание. Подчиненный может вызвать другие подчиненные функции, причем начальник не будет осведомлен об этом. Далее мы увидим, как это «скрытие» деталей выполнения способствует разработке хорошего программного обеспечения. На рис. 1 показана функция **main**, иерархически связанная с несколькими рабочими функциями. Заметим, что **workerl** действует как функция начальника по отношению к **worker4** и **worker5.** Взаимоотношения между функциями могут отличаться от показанных на этом рисунке иерархических отношений.

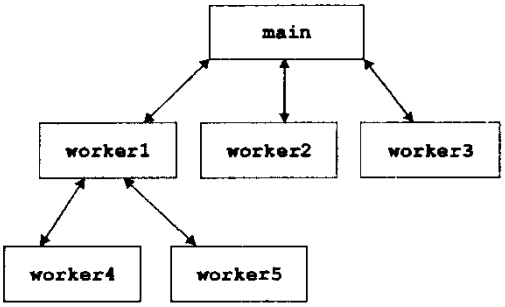

Рисунок 1 - Иерархическое взаимоотношение функция-начальник / функция-подчиненный

Математические библиотечные функции. Математические библиотечные функции позволяют программисту выполнять определенные типовые математические вычисления. Далее разнообразные математические библиотечные функции будут использоваться для иллюстрации самой концепции функций.

Обычно функция вызывается путем записи имени функции, после которого записывается левая круглая скобка, затем *аргумент* функции (или список аргументов, разделенных запятыми), а завершает запись правая круглая скобка. Например, программист, желающий вычислить и напечатать квадратный корень из 900.0, мог бы написать

```
printf("Koren' kvadratni = %.2f", sqrt(900.0));
```

При выполнении этого оператора вызывается математическая библиотечная функция **sqrt**, которая вычисляет квадратный корень из числа, заключенного в круглые скобки (900.0). Число 900.0 является аргументом функции **sqrt**. Приведенный выше оператор должен был бы напечатать 30.00. Функция **sqrt** получает аргумент типа **double** и возвращает результат типа **double**. Вообще все функции в математической библиотеке возвращают данные типа **double.**

При использовании функций математической библиотеки включайте соответствующий заголовочный файл с помощью директивы препроцессора **#include**<math.h>.

Аргументами функции могут быть константы, переменные или выражения. Если cl = 13.0, d = 3.0 и f = 4.0, то оператор

```
printf("Koren' kvadratni = %.2f", sqrt(cl + d*f));
```

вычислит и напечатает квадратный корень из 13.0 + 3.0 * 4.0 = 25.0, а именно, напечатает 5.00. Некоторые математические библиотечные функции приведены в таблице 1, где переменные x и y имеют тип double.

Таблица 1. Наиболее употребительные математические библиотечные функции

Функция	Описание	Пример
sqrt(x)	корень квадратный из x	sqrt(900.0) = 30.0 sqrt(9.0) = 3.0
exp(x)	экспоненциальная функция e^x	exp(1.0) = 2.718282 exp(2.0) = 7.389056
log(x)	логарифм натуральный x (по основанию e)	log(2.718282) = 1.0 log(7.389056) = 2.0
log10(x)	логарифм десятичный x (по основанию 10)	log10(1.0) = 0.0 log10(10.0) = 1.0 log10(100.0) = 2.0
fabs(x)	абсолютное значение x	если x > 0, то fabs(x) = x если x = 0, то fabs(x) =0.0 если x < 0, то fabs(x) = -x
ceil(x)	округление x до наименьшего целого, не меньшего чем x	ceil(9.2) = 10.0 ceil(-9.8) = -9.0
floor(x)	округление x до наибольшего целого, не большего чем x	floor(9.2) = 9.0 floor(-9.8) = -10.0
pow(x, y)	x в степени y	pow(2, 7) = 128.0 pow(9,0.5) = 3.0
fmod(x, y)	остаток от x/y, как число с плавающей точкой	fmod(13.657, 2.333) = 1.992
sin(x)	синус x (x в радианах)	sin(0.0) = 0.0
cos(x)	косинус x (x в радианах)	cos(0.0) = 1.0
tan(x)	тангенс x (x в радианах)	tan(0.0) = 0.0

Функции. Функции позволяют пользователю использовать модульное программирование (составлять программу из модулей). Все переменные объявляются в описаниях функций *локальными переменными* — они известны только для функции, в которой они описаны. Большинство функций имеют список *параметров,* который обеспечивает значения для связующей информации между функциями. Параметры тоже являются локальными переменными.

В программах, содержащих много функций, **main** должна быть построена как группа вызовов функций, которые и выполняют основную часть работы.

Существует несколько причин для построения программ на основе функций. Подход «разделяй и властвуй» делает разработку программ более управляемой. Другая причина — *повторное использование программных кодов,* т.е. использование существующих функций как стандартных блоков для создания новых программ. Повторное использование — основной фактор развития объектно-ориентированного программирования. При продуманном присвоении имен функций и хорошем их описании программа может быть создана быстрее из стандартизированных функций, соответствующих определенным задачам. Третья причина — желание избежать в программе повторения каких-то фрагментов. Код, оформленный в виде функции, может быть выполнен в разных местах программы простым вызовом этой функции.

Определения функций. Каждая программа, которую мы рассматривали, содержала функцию, называемую **main**, которая вызывает стандартные библиотечные функции для выполнения соответствующих им задач. Теперь мы рассмотрим, как программисты пишут свои собственные необходимые им функции. Рассмотрим программу, которая использует функцию **square** для вычисления квадратов целых чисел от 1 до 10 (пример 1). Функция **square** *активизируется* или *вызывается* в main вызовом **square(x)**.

Функция создает копию значения x в *параметре* y. Затем square вычисляет y * y. Результат передается в ту точку **main**, из которой была вызвана **square**, и затем этот результат выводится на экран. Благодаря структуре повторения **for** этот процесс повторяется десять раз.

Описание **square** показывает, что эта функция ожидает передачи в нее целого параметра y. Ключевое слово **int**, предшествующее имени функции, указывает, что **square** возвращает целый результат. Оператор **return** в **square** передает результат вычислений обратно в вызывающую функцию.

Пример 1:.

```
//  функция square, определяемая  программистом
#include <stdio.h>
int  square(int);      //  прототип функции

int main()
{
      for (int x = 1; x <= 10; x++)
        printf(" %d\n ", square(x));
    return 0;
}

//  описание функции
int square(int  y)
{ return y * y; }
```

Задание 1: Выполнить пример 1, приведенный в методических рекомендациях.

Строка

```
int square(int);
```

является *прототипом функции.* Тип данных **int** в круглых скобках указывает компилятору, что функция **square** ожидает в операторе вызова целое значение аргумента. Тип данных **int** слева от имени функции **square** указывает компилятору, что **square** возвращает оператору вызова целый результат. Компилятор обращается к прототипу функции для проверки того, что вызовы функции **square** содержат правильный возвращаемый тип, правильное количество аргументов, правильный тип аргументов и правильный порядок перечисления аргументов.

Формат описания функции имеет вид

```
тип-возвращаемого-значения имя-функции   (список-параметров)
{
        объявления  и  операторы;
}
```

Имя-функции — это любой правильно написанный идентификатор. *Тип-возвращаемого-значения* — это тип данных результата, возвращаемого из функции оператору ее вызова. Тип возвращаемого значения **void** указывает, что функция не возвращает никакого значения. Компилятор предполагает тип **int** для неопределенного типа возвращаемого значения.

Пропуск типа возвращаемого значении в описании функции вызывает синтаксическую ошибку, если прототип функции определяет возвращаемый тип иначе, чем **int.**

Если забыть вернуть значение из функции, в которой предполагается возвращение результата, это может привести к неожиданным ошибкам. Описание языка С указывает, что результат такой оплошности не определенен. В этом случае обычно компиляторы С выдают предупреждающее сообщение.

Возвращение какого-то значения из функции, для которой тип возвращаемого значения объявлен как **void**, вызывает синтаксическую ошибку.

Несмотря на то, что пропущенный тип возвращаемого значения по умолчанию **int**, всегда задавайте тип возвращаемого значения явным образом. Исключением является функция **main**, для которой тип возвращаемого значения обычно не указывается.

Список-параметров — это список разделенных запятыми объявлений тех параметров, которые получает функция при ее вызове. Если функция не получает никаких значений, *список-параметров* задается как **void**. Тип должен быть указан явно для каждого параметра в списке параметров.

Объявление параметров функции, имеющих одинаковый тип, в виде **float x, y** вместо **float x, float y**. Объявление параметра **float x, y** вызовет ошибку компиляции, так как типы надо указывать для каждого параметра в списке.

Повторное определение параметра функции как локальной переменной этой функции является синтаксической ошибкой.

Не используйте одинаковые имена для аргументов, передаваемых в функцию, и соответствующих параметров в описании функции, хотя это и не является ошибкой. Использование разных имен помогает избежать двусмысленности.

Объявления и *операторы* внутри фигурных скобок образуют *тело функции.* Тело функции рассматривается как *блок.* Блок — это просто составной оператор, который включает объявления. Переменные могут быть объявлены в любом блоке, а блоки могут быть вложенными. *При любых обстоятельствах функция не может быть описана внутри другой функции.*

Описание функции внутри другой функции является синтаксической ошибкой.

Прототип функции, заголовок функции и вызовы функции должны быть

согласованы между собой по количеству, типу и порядку следования аргументов и параметров и по типу возвращаемых результатов.

Существует три способа возврата управления к точке, из которой была вызвана функция. Если функция не должна возвращать результат, управление возвращается или просто при достижении правой фигурной скобки, завершающей функцию, или при выполнении оператора

return;

Если функция должна возвращать результат, то оператор

return *выражение;*

возвращает значение *выражения* в обращение к функции.

Второй пример использует определенную программистом функцию **maximum** для определения и возвращения наибольшего из трех целых чисел. Вводятся три целых числа. Затем эти целые числа передаются функции maximum, которая определяет наибольшее из чисел. Это значение возвращается функции **main** с помощью оператора return в **maximum** и печатается.

Пример 2:.

```
// Определение максимального из трех целых чисел
#include <stdio.h>
int maximum(int, int, int); // прототип функции

int main()
{
      int a, b, c;
      printf("Vvedite 3 chisla: ");
      scanf("%d%d%d", &a, &b, &c);
      printf("Maxsimum =  %d", maximum(a, b, c));
      return 0;
}

// Определение функции maximum
int maximum(int x, int y, int z)
{
      int max=x;
      if (y > max) max = y;
      if (z > max) max = z;
      return max;
}
```

Прототипы функций. *Прототип функции* является одной из наиболее важных особенностей С. Прототип функции указывает компилятору тип данных, возвращаемых функцией, количество параметров, которое ожидает функция, тип параметров и ожидаемый порядок их следования. Компилятор использует прототип функции для проверки правильности вызовов функции. Ранние версии С не выполняли такого рода проверку, поэтому был возможен неправильный вызов функции без обнаружения ошибок компилятором. Подобные вызовы приводили к неисправимым ошибкам выполнения или к хитроумным исправимым логическим ошибкам, которые было очень трудно обнаружить. Прототипы функции устранили этот недостаток.

Прототип функции maximum в примере 2 имеет вид:

int maximum(**int**, **int**, **int**);

Этот прототип указывает, что maximum имеет три аргумента типа int и возвращает результат типа int. Заметим, что прототип функции такой же, как заголовок описания

функции maximum, за исключением того, что в него не включены имена параметров (x, y и z).

Отсутствие точки с запятой в конце прототипа функции является синтаксической ошибкой.

Вызов функции, который не соответствует прототипу функции, ведет к синтаксической ошибке. Синтаксическая ошибка возникает также в случае отсутствия согласования между прототипом и описанием функции. Например, если бы в примере2 прототип функции был бы написан так:

```
void maximum(int, int, int);
```

компилятор сообщил бы об ошибке, потому что возвращаемый тип **void** в прототипе функции отличался бы от возвращаемого типа **int** в заголовке функции.

Другой важной особенностью прототипов функций является *приведение типов аргументов,* т.е. задание аргументам подходящего типа. Например, математическая библиотечная функция **sqrt** может быть вызвана с аргументом целого типа, даже если функция прототип в **math.h** определяет аргумент типа **double**, и при этом функция будет работать правильно. Оператор

```
printf("Koren' kvadratni = %.2f", sqrt(4));
```

правильно вычисляет sqrt(4), и печатает значение 2.00. Прототип функции заставляет компилятор преобразовать целое значение 4 в значение 4.0 типа **double** прежде, чем значение будет передано в **sqrt**. Вообще, значения аргументов, которые первоначально не соответствуют типам параметров в прототипе функции, преобразуются в подходящий тип перед вызовом функции. Эти преобразования могут привести к неверным результатам, если не руководствоваться *правилами приведения типов* С. Правила приведения определяют, как типы могут быть преобразованы в другие типы без потерь. В приведенном выше примере **sqrt** тип **int** автоматически преобразуется в **double** без изменения его значений. Однако **double** преобразуется в **int** с отбрасыванием дробной части значения **double**. Преобразование больших целых типов в малые целые типы (например, **long** в **short**) может также привести к изменению значений.

Отсутствие прототипа функции, когда функция не определена перед ее первым вызовом, приводит к синтаксической ошибке.

Прототип функции, размещенный вне описания какой-то другой функции, относится ко всем вызовам данной функции, появляющимся после этого прототипа в данном файле. Прототип функции, размещенный внутри описания некоторой функции, относится только к вызовам внутри этой функции.

Пример 3: Пример программы, которая выводит на экран меню и предлагает пользователю сделать выбор. В зависимости от выбора пользователя программа либо подсчитывает сумму двух чисел либо подсчитывает длину введенной строки. В программе кроме главной функции создано еще три функции.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int menu(); // Прототип функции menu.
int sum(int aa, int bb); // Прототип функции sum
void kol(char *s); // Прототип функции kol

int main()
{   int a,b,c=0,k1=0;
    char s1[20];
    while(1)
    {     //вызов функции menu
```

```
            switch(menu()) //условное выражение в операторе switch -
            {           //это значение, которое возвращет функция menu.
            case 1:
                  printf("vvedite a i b: ");
                  scanf("%d%d",&a,&b);
                  c=sum(a,b);             //вызов функции sum
                  printf("\nSum= %d\n",c);
                  break;
            case 2:
                  fflush(stdin);
                  printf("vvedite ctroku: \n");
                  gets(s1);
                  kol(s1);      //вызов функции kol
                  break;
            case 3: return 0;
          //Выполнение функции завершается после выполнения оператора return
            }
      }
   }
   /*Функция menu выводит на экран 5 строк меню, и считывает введенное
пользователем значение. Никаких параметров функции не передается, функция
возвращает целочисленное значение - введеное пользователем значение */
   int menu()  //определение функции menu
   {
      int ch;
      do {
            printf("\n   Menu: \n");
            printf("1. Podschitat' summu chisel:  \n");
            printf("2. Podschitat' kolichestvo simvolov v stroke:  \n");
            printf("3. Exit\n");
            printf("\t Vash vibor: ");
            scanf("%d",&ch);
      }while(ch>3);
      return ch;  // функция возвращает значение переменной ch, т.е. введенное
число.
   }

   /*Функция sum подсчитвывает сумму двух чисел. Функции передаются 2
целочисленных параметра - складываемые числа, функция возвращает
целочисленный параметр - сумму двух чискл.  */
   int sum(int aa, int bb) //определение функции sum
   {   return aa+bb;
   //Функция возвращает сумму двух чисел, которые передаются параметрами
функции
   }

   /*Функция kol подсчитвывает и выводит на экран длину строки. Функции
передается
   введенная пользователем строка, функция ничего не возвращает.  */
   void kol(char *s) //определение функции kol
   {   printf("Dlina=%d", strlen(s));
      // оператор return отсутствует, т.к. функция не возвращает значений
   }
```

Передача массивов в функцию. Если в качестве передаваемого аргумента используется массив данных, то в функцию передается *только указатель на массив*, т.е. адрес первого элемента. При вызове функции в списке аргументов записывается имя массива. Имя массива без индекса является адресом элемента с нулевым индексом.

Можно использовать *три варианта описания массива*, поступающего в функцию, в качестве параметра.

1. Параметр в функции может быть объявлен как массив соответствующего типа с *указанием его размера*.

Пример 4:Элементы матрицы равны произведению номера столбца на номер строки. Подсчитать сумму элементов матрицы.

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

int sum(int x[5])       // определение функции, размер указан
{     int res=0;              // int sum(int x[]);        //  возможные
      for(int i=0; i<5; i++)  // int sum(int *);            //  варианты
              res+=x[i];      // int sum(int []);     //  прототипа
    return res;
}

int main()
{
      int mas[3][5], i, j;
      for(i=0; i<3; i++)
            for(j=0; j<5; j++)
                  mas[i][j]=i*j;
      for(i=0; i<3; i++)
            printf("\n res=%d\n", sum(mas[i]));
      for(i=0; i<3; i++)
      {     for(j=0; j<5; j++)
                  printf("%d ", mas[i][j]);
          printf("\n");
      }
}
```

В этом случае Си автоматически преобразует mas[i] к указателю на целый тип, и в функцию передается адрес первого элемента каждой строки матрицы, т.е. адрес элемента mas[i][0]. В функции вычисляется сумма элементов каждой строки матрицы и сумма выводится на экран.

2. Массив в качестве параметра в функции может быть объявлен *без указания его размера*. Так как сам массив в стек не копируется, то размер массива в общем случае компилятору не требуется. Контроль правильности использования индекса массива возлагается на программиста. В этом случае также используется указатель на первый элемент массива.

```
int sum(int x[])                    // определение функции, размер не указан
   {int res=0;
        for(int i=0; i<5; i++)  //    i<5 - известно программисту
              res+=x[i];
        return res;
    }                                   //   Вызов функции остается тем же
```

3. Наиболее распространенный способ, обычно используемый при написании профессиональных программ – объявление параметра-массива *указателем* на соответствующий тип данных. Функцию можно вызывать так, как было проиллюстрировано выше, но можно и так:

```
int sum(int *x)                    // определение функции
   { int res=0;
     for(int i=0; i<5; i++)
              res+=*x++;                  // можно и res+=x[i]
     return res;
    }
```

Во всех случаях в функцию передается адрес первого элемента массива, т.е. указатель соответствующего типа.

Пример 5: Ввести матрицу n x m. Написать функцию, которая позволяет ввести матрицу и вывести ее на экран.

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

int x[50][50];
void vvod(int x[50][50], int n1,int m1);
int vivod(int x[50][50], int n1, int m1);

int main()
{     int n,m;
      printf("vvedite n\n");
      scanf("%d",&n);
      printf("vvedite m\n");
      scanf("%d",&m);
      vvod(x, n, m);
      vivod(x, n, m);
}
void vvod(int x[50][50], int n1, int m1)
{     int z,j;
      for (z=0;z<n1;z++)
            for (j=0;j<m1;j++){
                  printf("Vvedite element [%d,%d]\n", z+1, j+1);
                  scanf("%d",&x[z][j]);
            }
}
int vivod(int x[50][50], int n1, int m1)
{     int z,j;
      printf("\n");
      for (z=0;z<n1;z++){
            for (j=0;j<m1;j++)
                  printf("%d ", x[z][j]);
            printf("\n");
      }
      return 0;
}
```

Практическая часть

Упражнение 1

Сделать все операции над матрицей в функции.

1. Найти в матрице первую строку, все элементы которой положительны, и сумму этих элементов. Уменьшить все элементы матрицы на эту сумму.
2. Найти в матрице первую строку, все элементы которой отрицательны. Увеличить все элементы матрицы на значение первого элемента найденной строки.
3. Найти в матрице первую строку, все элементы которой упорядочены по возрастанию. Изменить упорядоченность элементов этой строки на обратную.
4. Найти в матрице первую строку, все элементы которой упорядочены по убыванию. Изменить упорядоченность элементов этой строки на обратную.
5. Проверить, есть ли в матрице хотя бы одна строка, содержащая положительный элемент, и найти ее номер.
6. Проверить, есть ли в матрице хотя бы одна строка, содержащая отрицательный элемент, и найти ее номер. Все
7. Найти в матрице первую строку, все элементы которой равны нулю.
8. Найти в матрице первую строку, все элементы которой отрицательны.
9. Найти в матрице первую строку, все элементы которой положительны, и сумму этих элементов. Уменьшить все элементы матрицы на эту сумму.
10. Найти в матрице первую строку, все элементы которой упорядочены по убыванию. Изменить упорядоченность элементов этой строки на обратную.

Упражнение 2

1. Ввести строку и удалить из нее пробелы в начале и конце строки. Полученную строку вывести.
2. Ввести строку и удалить из нее лишние пробелы. Полученную строку вывести.
3. Ввести строку. В функции посчитать количество пробелов.
4. Ввести строку. В функции ввести символ и посчитать количество его вхождения в строку.
5. Ввести строку. Найти в функции в ней слово максимальной длины.
6. Ввести строку. Найти в функции в ней слово минимальной длины.
7. Ввести 2 строки. Выполнить в функции замену каждого символа из 2 –ой строки который встречается в 1-ой на 0.
8. Ввести строку. В функции ввести символ и посчитать количество его вхождения в строку.
9. Ввести строку и удалить из нее лишние пробелы. Полученную строку вывести.
10. Ввести строку. Найти в функции в ней слово минимальной длины.