2.7. Функции пользователя

С увеличением объема программы ее код становится все более сложным. Одним из способов борьбы со сложностью любой задачи является ее разбиение на части. В языке Си, как и в любом языке программирования высокого уровня, задача может быть разбита на более простые подзадачи при помощи подпрограмм-функций. После этого программу можно рассматривать в более укрупненном виде – на уровне взаимодействия созданных подпрограмм. Использование подпрограмм в коде программы и ведет к упрощению ее структуры.

Разделение программы на подзадачи позволяет также избежать избыточности кода, поскольку функцию записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы. Кроме того, упрощается процесс отладки программы, содержащей подпрограммы. Часто используемые функции можно помещать в отдельные библиотеки.

Следующим шагом в повышении уровня абстракции программы является группировка функций и связанных с ними данных в отдельные файлы (модули), компилируемые раздельно. Получившиеся в результате компиляции объектные модули объединяются в исполняемую программу с помощью компоновщика. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля, что позволяет отлаживать программу по частям.

Для того чтобы использовать модуль, достаточно знать только его интерфейс, а не все детали его реализации.

Разделение программы на максимально обособленные части (подпрограммы) является довольно сложной задачей, которая должна решаться на этапе проектирования программы.

В отличие от других языков программирования высокого уровня в языке Си нет разделения на подпрограммы-процедуры и подпрограммы-функции, здесь вся программа строится только из функций.

В языке Си любая подпрограмма является функцией, представляющей собой отдельный программный модуль, к которому можно обратиться, чтобы передать через параметры исходные данные и получить один или несколько результатов его работы.

Функция – это именованная последовательность инструкций, выполняющая какое-либо законченное действие.

Таким образом, любая программа на языке Си состоит из функций. Минимальная программа на Си содержит, как уже известно, единственную функцию *main* (основная, главная), с которой и начинается выполнение программы.

Декларация функции

Как и любой объект программы на языке Си, пользовательские функции необходимо декларировать. Объявление функции пользователя, т.е. ее декларация, выполняется в двух формах – в форме описания (объявления) и в форме определения, т.е. любая пользовательская функция должна быть объявлена и определена.

Описанием функции является декларация ее прототипа, который сообщает компилятору о том, что далее будет приведено ее полное определение (текст), т.е. реализация.

Объявление функции (прототип, заголовок) задает ее свойства – идентификатор, тип возвращаемого значения (если такое имеется), количество и типы параметров.

В стандарте языка используется следующий формат декларации (объявления) функций:

***тип_результата ID_функции* (*список*);**

В *списке* перечисляются типы параметров данной функции, причем идентификаторы переменных в круглых скобках прототипа указывать необязательно, т.к. компилятор языка их не обрабатывает.

Описание прототипа дает возможность компилятору проверить соответствие типов и количества параметров при фактическом вызове этой функции.

Пример объявления функции *fun*, которая имеет три параметра типа *int*, один параметр типа *double* и возвращает результат типа *double*:

```
double fun(int, int, int, double);
```

Каждая функция, вызываемая в программе, должна быть определена (только один раз). ***Определение функции*** – это ее полный текст, включающий заголовок и код.

Полное определение (реализация) функции имеет следующий вид:

```
тип_результата  ID_функции(список параметров)
                {
                        код функции
                    return  выражение;
                }
```

Рассмотрим составные части определения пользовательской функции.

Тип результата определяет тип *выражения*, значение которого возвращается в точку ее вызова при помощи оператора ***return*** *выражение*; (возврат). *Выражение* преобразуется к *типу_результата*, указанному в заголовке функции и передается в точку вызова. Тип возвращаемого функцией значения может быть любым базовым типом, а также указателем на массив или функцию. Если функция не должна возвращать значение, указывается тип ***void***. В данном случае оператор *return* можно не ставить. Из функции, которая не описана как

void, необходимо возвращать значение, используя оператор *return*. Если тип функции не указан, то по умолчанию устанавливается тип *int*.

Список параметров состоит из перечня типов и идентификаторов параметров, разделенных запятыми. Список параметров определяет объекты, которые требуется передать в функцию при ее вызове.

В определении и в объявлении одной и той же функции типы и порядок следования параметров должны совпадать. Тип возвращаемого значения и типы параметров совместно определяют тип функции.

Функция может не иметь параметров, но круглые скобки необходимы в любом случае. Если у функции отсутствует список параметров, то при декларации такой функции желательно в круглых скобках указать *void*. Например, *void main*(*void*){ ... }.

В функции может быть несколько операторов *return*, но может и не быть ни одного (тип *void* – это определяется потребностями алгоритма). В последнем случае возврат в вызывающую программу происходит после выполнения последнего оператора кода функции.

Пример функции, определяющей наименьшее значение из двух целочисленных переменных:

```
int  min (int x, int y)
{
        return (x<y) ? x : y;
}
```

Функции, возвращающие значение, желательно использовать в правой части выражений языка Си, иначе возвращаемый результат будет утерян.

В языке Си каждая функция – это отдельный блок программы, вход в который возможен только через вызов данной функции.

Вызов функции

Для вызова функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечислить список передаваемых ей ***аргументов***. Вызов функции может находиться в любом месте программы, где по синтаксису допустимо выражение того типа, который формирует функция.

Простейший вызов функции имеет следующий формат:

ID_функции (***список аргументов***);

где в качестве аргументов можно использовать константы, переменные, выражения (их значения перед вызовом функции будут определены компилятором).

Аргументы в списке вызова должны совпадать со списком параметров вызываемой функции по количеству и порядку следования, а типы аргументов при передаче в функцию будут преобразованы, если это возможно, к типу соответствующих им параметров.

Связь между функциями осуществляется через аргументы и возвращаемые функциями значения. Ее можно осуществить также через внешние, глобальные переменные (см. гл. 12).

Глобальные переменные доступны всем функциям, где они не описаны как локальные переменные. Использовать их для передачи данных между функциями довольно просто, но тем не менее этого делать не рекомендуется. Необходимо стремиться к тому, чтобы функции в программе были максимально независимыми и чтобы их интерфейс полностью определялся прототипами этих функций.

Функции могут располагаться в исходном файле в любом порядке, при этом исходная программа может размещаться в нескольких файлах.

Все величины, описанные внутри функции, являются локальными. Областью их действия является функция. При вызове функции, как и при входе в любой блок, в стеке выделяется память под локальные автоматические переменные. Кроме того, в стеке сохраняется содержимое регистров процессора на момент, предшествующий вызову функции, и адрес возврата из функции, для того чтобы при выходе из нее можно было продолжить выполнение вызывающей функции. При выходе из функции соответствующий участок стека освобождается, поэтому значения локальных переменных между вызовами одной и той же функции не сохраняются. Если этого требуется избежать, при объявлении локальных переменных используется модификатор *static*, например:

```
#include <stdio.h>
        void f1(int);
void main(void)
{
        f1(5);
}
void f1(int i)
{
        int m=0;
        puts(" n  m  p ");
        while (i--) {
                static int n = 0;
                int p = 0;
                printf(" %d  %d  %d \n", n++ , m++ , p++);
        }
}
```

Статическая переменная *n* будет создана в сегменте данных ОП и проинициализируется нулем только один раз при первом выполнении оператора, содержащего ее определение, т.е. при первом вызове функции *f*1. Автоматическая переменная *m* инициализируется при каждом входе в функцию. Автоматическая переменная *p* инициализируется при каждом входе в блок цикла.

В результате выполнения программы получим

n m p

0 0 0
1 1 0
2 2 0
3 3 0
4 4 0

Передача аргументов в функцию

В языке Си аргументы при стандартном вызове функции передаются по значению. Это означает, что в стеке, как и в случае локальных данных, выделяется место для формальных параметров функции. В выделенное место при вызове функции заносятся значения фактических аргументов, при этом проверяется соответствие типов и при необходимости выполняются их преобразования. При несоответствии типов выдается диагностическое сообщение. Затем функция использует и может изменять эти значения в стеке.

При выходе из функции измененные значения теряются, т.к. время жизни и зона видимости локальных параметров определяется кодом функции. Вызванная функция не может изменить значения переменных, указанных как фактические аргументы при обращении к данной функции.

В случае необходимости функцию можно использовать для изменения передаваемых ей аргументов. В этом случае в качестве аргумента необходимо в вызываемую функцию передавать не значение переменной, а ее адрес.

При передаче по адресу в стек заносятся копии адресов аргументов, а функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения аргументов. Для обращения к значению аргумента-оригинала используется операция «*».

Пример функции, в которой меняются местами значения *x* и *y*:

```
void  zam(int *x, int *y)
{
        int  t = *x;
        *x = *y;
        *y = t;
}
```

Участок программы с обращением к данной функции:

```
void zam (int*, int*);
void main (void)
{
        int a=2, b=3;
        printf(" a = %d , b = %d\n", a, b);
        zam (&a, &b);
        printf(" a = %d , b = %d\n", a, b);
}
```

При таком способе передачи данных в функцию их значения будут изменены, т.е. на экран монитора будет выведено

a = 2 , b=3
a = 3 , b=2

Если требуется запретить изменение значений какого-либо параметра внутри функции, то в его декларации используют атрибут *const*, например:

void f1(int, const double);

Рекомендуется указывать *const* перед всеми параметрами, изменение которых в функции не предусмотрено. Это облегчает, например, отладку программы, т.к. по заголовку функции видно, какие данные в функции изменяются, а какие нет.

Операция typedef

Любому типу данных, как стандартному, так и определенному пользователем, можно задать новое имя с помощью операции *typedef*:

typedef тип новое_имя **;**

Введенный таким образом новый тип используется аналогично стандартным типам, например, введя пользовательские типы:

typedef unsigned int ***UINT***; – здесь *UINT* – новое имя;
typedef char ***M_s*** [101]; – здесь *M_s* – тип пользователя, определяющий строки, длиной не более 100 символов.

Декларации объектов введенных типов будут иметь вид

UINT i, j; → две переменные типа *unsigned int* ;
M_s str[10]; → массив из 10 элементов, в каждом из которых можно хранить по 100 символов.

Рассмотренная операция упростит использование указателей на функции, которые рассматриваются в следующем разделе.

Указатели на функции

В языке Си идентификатор функции является константным указателем на начало функции в оперативной памяти и не может быть значением переменной. Но имеется возможность декларировать указатели на функции, с которыми можно обращаться как с переменными (например, можно создать массив, элементами которого будут указатели на функции).

Рассмотрим методику работы с указателями на функции.

1. Как и любой объект языка Си, указатель на функции необходимо декларировать. Формат объявления указателя на функции следующий:

тип (****переменная-указатель***)(***список параметров***);

т.е. декларируется указатель, который можно устанавливать на функции, возвращающие результат указанного *типа* и которые имеют указанный *список параметров*. Наличие первых круглых скобок обязательно, так как без них – это декларация функции, которая возвращает указатель на результат.

Например, объявление вида

double (*__p_f__)(char, double);

говорит о том, что декларируется указатель *p_f*, который можно устанавливать на функции, возвращающие результат типа *double* и имеющие два параметра: первый – символьного типа, а второй – вещественного типа.

2. Идентификатор функции является константным указателем, поэтому для того чтобы установить переменную-указатель на конкретную функцию, достаточно ей присвоить ее идентификатор:

переменная-указатель = ID_функции;

Например, имеется функция с прототипом: double f1(char, double); тогда операция

p_f = f1;

установит указатель *p_f* на данную функцию.

3. Вызов функции после установки на нее указателя выглядит так:

(*переменная-указатель)(список аргументов);

или

переменная-указатель (список аргументов);

После таких действий кроме стандартного обращения к функции:

ID_функции(список аргументов);

появляется еще два способа вызова функции:

(*переменная-указатель)(список аргументов);

или

переменная-указатель (список аргументов);

Последняя запись справедлива, так как *p_f* также является адресом начала функции в оперативной памяти.

Для нашего примера к функции *f*1 можно обратиться следующими способами:

f1(‘z’, 1.5); – обращение к функции по имени (ID);
(* p_f)(‘z’, 1.5); – обращение к функции по указателю;
p_f(‘z’, 1.5); – обращение к функции по ID указателя.

Основное назначение указателей на функции – это обеспечение возможности передачи идентификаторов функций в качестве параметров в функцию, которая реализует некоторый вычислительный процесс, используя формальное имя вызываемой функции.

Пример: написать функцию вычисления суммы *sum*, обозначив слагаемое формальной функцией *fun*(*x*). При вызове функции суммирования передавать через параметр реальное имя функции, в которой задан явный вид слагаемого. Например, пусть требуется вычислить две суммы:

$$S_1 = \sum_{i=1}^{2n} \frac{x}{5} \qquad \text{и} \qquad S_2 = \sum_{i=1}^{n} \frac{x}{2} \ .$$

Поместим слагаемые этих сумм в пользовательские функции *f*1 и *f*2 соответственно. При этом воспользуемся операцией *typedef*, введя пользовательский тип данных: указатель на функции ***p_f***, который можно устанавливать на функции, возвращающие результат *double* и имеющие один параметр типа *double*.

Тогда в списке параметров функции суммирования достаточно будет указать фактические идентификаторы функций созданного типа ***p_f***.

Текст программы для решения данной задачи может быть следующим:

```
. . .
typedef double (*p_f)(double);
double sum(p_f, int, double);          // Декларации прототипов функций
double f1(double);
double f2(double);
void main(void)
{
    double x, s1, s2;
    int n;
    puts (" Введите кол-во слагаемых n и значение x: ");
    scanf (" %d %lf ", &n, &x);
    s1 = sum (f1, 2*n, x);
    s2 = sum (f2, n, x);
    printf("\n\t N = %d , X = %lf ", n, x);
    printf("\n\t Сумма 1 = %lf\n\t Сумма 2 = %lf ", s1, s2);
}
/*  Первый параметр функции суммирования – формальное имя функции, введенное с помощью typedef типа */
double sum(p_f  fun, int n, double x)  {
    double s=0;
    for(int i=1; i<=n; i++)
        s+=fun(x);
    return s;
}
//————————— Первое слагаемое ———————————
double f1(double r) {
    return r/5.;
}
//————————— Второе слагаемое ———————————
double f2(double r) {
    return r/2.;
}
```

В заключение рассмотрим оптимальную передачу в функции одномерных и двухмерных массивов.

Передача в функцию одномерного массива:

```
void main(void)
{
        int vect[20];
                …
        fun(vect);
                …
 }
void   fun( int v[ ]) {
        …
}
```

При использовании в качестве параметра одномерного массива в функцию передается указатель на его первый элемент, т.е. массив всегда передается по адресу и параметр v[ ] преобразуется в *v. Поэтому этой особенностью можно воспользоваться сразу:

```
void   fun( int *v) {
        …
}
```

При этом информация о количестве элементов массива теряется, так как размер одномерного массива недоступен вызываемой функции. Данную особенность можно обойти несколькими способами. Например, передавать его размер через отдельный параметр. Если же размер массива является константой, можно указать ее и при описании формального параметра, и в качестве границы циклов при обработке массива внутри функции:

```
 void   fun( int v[20]) {
        . . .
}
```

В случае передачи массива символов, т.е. строки, ее фактическую длину можно определить по положению признака окончания строки (нуль-символа) через стандартную функцию *strlen*.

Передача в функцию двухмерного массива:

Если размеры известны на этапе компиляции, то

```
void f1(int m[3][4])  {
        int i, j;
        for ( i = 0; i<3; i++)
                for ( j = 0; j<4; j++)
                        . . .                                   // Обработка массива
        }
```

Двухмерный массив, как и одномерный, также передается как указатель, а указанные размеры используются просто для удобства записи. При этом первый размер массива не используется при поиске положения элемента массива в ОП, поэтому передать массив можно так:

```
void main(void)
{
        int mas [3][3]={{1,2,3}, {4,5,6}};
                …
        fun (mas);
                …
}
void fun( int m[ ][3]) {
        …
}
```

Если же размеры двухмерного массива, например, вводятся с клавиатуры (неизвестны на этапе компиляции), то их значения следует передавать через дополнительные параметры, например:

```
        …
void fun( int**, int, int);
void main()
{
        int **mas, n, m;
                ...
        fun (mas, n, m);
                …
}
void fun( int **m, int n, int m)  {
                . . .                                   // Обработка массива
}
```

Рекурсивные функции

Рекурсивной (самовызываемой или самовызывающей) называют функцию, которая прямо или косвенно вызывает сама себя.

При каждом обращении к рекурсивной функции создается новый набор объектов автоматической памяти, локализованных в коде функции.

Возможность прямого или косвенного вызова позволяет различать прямую или косвенную рекурсии. Функция называется косвенно рекурсивной в том случае, если она содержит обращение к другой функции, содержащей прямой или косвенный вызов первой функции. В этом случае по тексту определения функции ее рекурсивность (косвенная) может быть не видна. Если в функции используется вызов этой же функции, то имеет место прямая рекурсия, т.е. функция по определению рекурсивная.

Рекурсивные алгоритмы эффективны в задачах, где рекурсия использована в самом определении обрабатываемых данных. Поэтому изучение рекурсивных методов нужно проводить, вводя динамические структуры данных с рекурсивной структурой. Рассмотрим вначале только принципиальные возможности, которые предоставляет язык Си для организации рекурсивных алгоритмов.

В рекурсивных функциях необходимо выполнять следующие правила.

1. При каждом вызове в функцию передавать модифицированные данные.

2. На каком-то шаге должен быть прекращен дальнейший вызов этой функции, это значит, что рекурсивный процесс должен шаг за шагом упрощать

задачу так, чтобы для нее появилось нерекурсивное решение, иначе функция будет вызывать себя бесконечно.

3. После завершения очередного обращения к рекурсивной функции в вызывающую функцию должен возвращаться некоторый результат для дальнейшего его использования.

***Пример* 1**. Заданы два числа *a* и *b*, большее из них разделить на меньшее, используя рекурсию.

Текст программы может быть следующим:

```
        . . .
double proc(double, double);
void main (void)
{
        double a,b;
        puts(“ Введи значения a, b : ”);
        scanf(“%lf %lf”, &a, &b);
        printf(“\n Результат деления : %lf”, proc(a,b));
}
//———————————— Функция ————————————————
double proc( double a, double b)   {
        if ( a< b ) return proc ( b, a );
                else  return a/b;
}
```

Если *a* больше *b*, условие, поставленное в функции, не выполняется и функция *proc* возвращает нерекурсивный результат.

Пусть теперь условие выполнилось, тогда функция *proc* обращается сама к себе, аргументы в вызове меняются местами и последующее обращение приводит к тому, что условие вновь не выполняется и функция возвращает нерекурсивный результат.

***Пример* 2**. Функция для вычисления факториала *неотрицательного* значения *k* (для возможных отрицательных значений необходимо добавить дополнительные условия).

```
double fact (int k) {
        if ( k < 1 ) return 1;
        else
                return  k * fact ( k – 1);
}
```

Для нулевого значения параметра функция возвращает 1 (0! = 1), в противном случае вызывается та же функция с уменьшенным на 1 значением параметра и результат умножается на текущее значение параметра. Тем самым для значения параметра *k* организуется вычисление произведения

$$k * (k–1) * (k–2) * ... * 3 * 2 * 1 * 1$$

Последнее значение «1» – результат выполнения условия $k < 1$ при $k = 0$, т.е. последовательность рекурсивных обращений к функции *fact* прекращается при вызове *fact*(0). Именно этот вызов приводит к последнему значению «1» в произведении, так как последнее выражение, из которого вызывается функция, имеет вид: 1 * *fact*(1 – 1).

***Пример* 3**. Рассмотрим функцию определения корня уравнения $f(x) = 0$ на отрезке [*a, b*] с заданной точностью eps. Предположим, что исходные данные задаются без ошибок, т.е. $eps > 0$, $f(a)*f(b) < 0$, $b > a$, и вопрос о возможности существования нескольких корней на отрезке [*a,b*] нас не интересует. Не очень эффективная рекурсивная функция для решения поставленной задачи приведена в следующей программе:

```
        . . .
int counter = 0;              // Счетчик обращений к тестовой функции
//————— Нахождение корня методом деления отрезка пополам ————————
double   Root(double f(double), double a, double b, double eps)  {
        double  fa = f(a), fb = f(b), c, fc;
        if ( fa * fb > 0) {
                printf("\n На интервале a,b НЕТ корня!");
                exit(1);
                }
        c = (a + b) / 2.0;
        fc = f(c);
        if (fc == 0.0 ||  fabs(b – a) < = eps) return c;
        return (fa * fc < 0.0) ? Root(f, a, c, eps) : Root(f, c, b, eps);
}
//————————————————————————————————————————————————
void main()
{
        double x, a=0.1, b=3.5, eps=5e–5;
        double fun(double) ;                    // Прототип тестовой функции
        x = Root (fun, a, b, eps) ;
        printf ("\n Число обращений к функции = %d . ", counter);
        printf ("\n Корень = %lf  . ", x);
}
//————————————— Определение тестовой функции  fun ———————————————
double fun (double x) {
        counter++;            // Счетчик обращений – глобальная переменная
        return (2.0/x * cos(x/2.0));
}
```

Значения *a, b* и *eps* заданы постоянными только для тестового анализа полученных результатов, хотя лучше данные вводить с клавиатуры.

В результате выполнения программы с определенными в ней конкретными данными получим:

Число обращений к функции = 54 .
Корень = 3.141601.

Неэффективность предложенной программы связана, например, с излишним количеством обращений к программной реализации функции, для которой определяется корень. При каждом рекурсивном вызове функции *Root* повторно вычисляются значения *f*(*a*) и *f*(*b*), хотя они уже известны после предыдущего вызова.

В литературе по программированию рекурсиям уделено достаточно внимания как в теоретическом плане, так и в плане рассмотрения механизмов реализации рекурсивных алгоритмов. Сравнивая рекурсию с итерационными методами, отмечают, что рекурсивные алгоритмы наиболее пригодны в случаях, когда поставленная задача или используемые данные определены рекурсивно. В тех случаях, когда вычисляемые значения определяются с помощью простых рекуррентных соотношений, гораздо эффективнее применять итерационные методы.

Таким образом, рассмотренные выше примеры только иллюстрируют схемы организации рекурсивных функций, но не являются примерами эффективного применения рекурсивного подхода к вычислениям.

При обработке динамических информационных структур, которые включают рекурсивность в само определение обрабатываемых данных, применение рекурсивных алгоритмов не имеет конкуренции со стороны итерационных методов.

Параметры командной строки функции main

В стандарте *ANSI* функция *main* возвращает целочисленный результат, т.е. используется следующим образом:

```
int main () {
        …
        return 0;
}
```

здесь оператор *return* возвращает операционной системе код завершения функции, причем значение 0 трактуется как нормальное завершение, остальные значения воспринимаются как ошибки.

Функция *main* может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию *main* используются два параметра, общепринятые (необязательные) идентификаторы которых *argc* и *argv*:

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

Параметр *argc* имеет тип *int*, его значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой функции. Параметр *argv* – это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ

пробел, то при записи его в командную строку оно должно быть заключено в кавычки.

Функция *main* может иметь и третий параметр *argp*, который служит для передачи параметров операционной системы (ОС), в которой выполняется программа, в этом случае ее заголовок имеет вид

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

Операционная система поддерживает передачу значений для параметров *argc, argv, argp*, а пользователь отвечает за передачу и использование фактических аргументов функции *main*.

Приведем пример программы печати фактических аргументов, передаваемых из ОС в функцию *main* и параметров оперативной системы.

```
int main ( int argc, char *argv[], char *argp[])
{
        int i;
        printf ("\n Program Name %s", argv[0]);
        for( i=1; i <=argc; i++)
            printf ("\n Argument %d  =  %s", i, argv[i]);
        printf ("\n ОС parametrs: ");
        while  (*argp) {
            printf ("\n %s", *argp);
            argp++;
        }
        return 0;
}
```

Очевидно, что оформленная таким образом функция *main*() может вызываться рекурсивно из любой функции.

Советы по программированию

При выполнении вариантов заданий придерживайтесь следующих ключевых моментов.

1. Размеры нединамических массивов задаются константами или константными выражениями. Рекомендуется для этого использовать поименованные константы.
2. Элементы массивов нумеруются с нуля, максимальный номер (индекс) элемента всегда на единицу меньше указанного размера.
3. Автоматический контроль выхода индексов элементов за указанные границы массива отсутствует.
4. Указатель – это переменная, в которой хранится адрес участка оперативной памяти.
5. Имя массива является указателем на его нулевой элемент, т.е. на его начало в оперативной памяти.
6. Обнуления динамической памяти при ее выделении не происходит. Инициализировать динамический массив нельзя.

7. Освобождение памяти, выделенной посредством стандартных функций, выполняется при помощи функции *free* (при использовании операции *new* – операцией *delete*).

8. Если количество элементов массива известно заранее, определяйте массив в области декларации переменных (причем лучше как локальный объект). Если же количество элементов массива можно задать во время выполнения программы, лучше создать динамический массив.

9. При задании длины строки необходимо учитывать завершающий нуль-символ (признак окончания строки).

10. Операция присваивания строк выполняется с помощью функции стандартной библиотеки.

11. Для ввода строк, содержащих пробелы, используют функцию *gets*.

12. Программа, написанная с использованием функций, может получиться более быстродействующей, но менее безопасной.

13. Недостатком символьных массивов является отсутствие проверки выхода за пределы отведенной им памяти.

14. Двухмерный массив хранится по строкам в непрерывной области памяти.

15. Первый индекс двухмерного массива трактуется как номер строки таблицы, второй – как номер столбца. Каждый индекс может изменяться от 0 до значения соответствующего размера, уменьшенного на единицу.

16. Для выделения динамической памяти под массив, в котором все размеры переменные, используются циклы.

17. Функция – это именованная последовательность операторов, выполняющая законченное действие. Функции используют для упрощения структуры программы.

18. Вызов функции осуществляется путем указания ее идентификатора (имени) и в случае необходимости – набора аргументов.

19. Передача аргументов в функцию может выполняться по значению или по адресу.

20. Массивы всегда передаются в функцию по адресу. Количество элементов в массиве должно передаваться отдельным параметром.

21. Рекурсивная функция должна содержать хотя бы одну не рекурсивную ветвь. При использовании рекурсии следует учитывать возникающее при этом использование дополнительной памяти.

ЗАДАНИЕ 4. Обработка массивов

Первый уровень сложности

Составить программу, решающую указанную ниже задачу.

В одномерном массиве, состоящем из *n* (не более 10) вводимых с клавиатуры значений, вычислить заданное значение.

1. Произведение элементов массива, расположенных между максимальным и минимальным элементами.

2. Сумму элементов массива, расположенных между первым и последним нулевыми элементами.

3. Сумму элементов массива, расположенных до последнего положительного элемента.

4. Сумму элементов массива, расположенных между первым и последним положительными элементами.

5. Произведение элементов массива, расположенных между первым и вторым нулевыми элементами.

6. Сумму элементов массива, расположенных между первым и вторым отрицательными элементами.

7. Сумму элементов массива, расположенных до минимального элемента.

8. Сумму целых частей элементов массива, расположенных после последнего отрицательного элемента.

9. Сумму элементов массива, расположенных после последнего элемента, равного нулю.

10. Сумму модулей элементов массива, расположенных после минимального по модулю элемента.

11. Сумму элементов массива, расположенных после минимального элемента.

12. Сумму элементов массива, расположенных после первого положительного элемента.

13. Сумму модулей элементов массива, расположенных после первого отрицательного элемента.

14. Сумму модулей элементов массива, расположенных после первого элемента, равного нулю.

15. Сумму положительных элементов массива, расположенных до максимального элемента.

Второй уровень сложности

Написать программу по обработке двухмерного массива. Предусмотреть динамический захват и освобождение памяти. Размеры массива n, m и значения элементов массива вводятся с клавиатуры.

1. Определить количество строк, не содержащих ни одного нулевого элемента.

2. Определить количество столбцов, не содержащих ни одного нулевого элемента.

3. Определить количество столбцов, содержащих хотя бы один нулевой элемент.

4. Определить произведение элементов в тех строках, которые не содержат отрицательных элементов.

5. Определить сумму элементов в тех столбцах, которые не содержат отрицательных элементов.

6. Определить сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

7. Найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

8. Найти сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.

9. Найти сумму модулей элементов, расположенных ниже главной диагонали.

10. Найти сумму модулей элементов, расположенных выше главной диагонали.

11. Найти количество строк, среднее арифметическое элементов которых меньше введенной с клавиатуры величины.

12. Найти номер первой из строк, содержащих хотя бы один положительный элемент.

13. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.

14. Найти номер первого из столбцов, не содержащих ни одного отрицательного элемента.

15. Найти номер первой из строк, не содержащих ни одного положительного элемента.

ЗАДАНИЕ 5. Функции пользователя

Первый уровень сложности

Составить программу из задания 3 (второй уровень сложности), в которой для каждого $\boldsymbol{x}$, изменяющегося от $\boldsymbol{a}$ до $\boldsymbol{b}$ с шагом $\boldsymbol{h}$, вычисление значений $Y(x)$ и $S(x)$ оформить в виде функций пользователя.

В основной функции реализовать следующие действия:

– ввод исходных значений a, b, h и n;

– обращение к функциям расчета $Y(x)$ и $S(x)$;

– вывод результатов в виде таблицы.

Если в задании используется значение факториала, его расчет также оформить функцией.

Второй уровень сложности

Решить поставленную задачу с использованием рекурсивной и обычной функций. Сравнить полученные результаты.

1. Для заданного целого десятичного числа N получить его представление в p-ичной системе счисления ($p < 10$).

2. В упорядоченном массиве целых чисел a_i (i=1, ..., n) найти номер находящегося в массиве элемента c, используя метод двоичного поиска.

3. Найти наибольший общий делитель чисел M и N, используя теорему Эйлера: если M делится на N, то $НОД(N, M) = N$, иначе $НОД(N, M) = (M \bmod N, N)$.

4. Числа Фибоначчи определяются следующим образом: $Fb(0) = 0$; $Fb(1) = 1$; $Fb(n) = Fb(n\text{-}1) + Fb(n\text{-}2)$. Определить $Fb(n)$.

5. Найти значение функции Аккермана $A(m, n)$, которая определяется для всех неотрицательных целых аргументов m и n следующим образом:

$A(0, n) = n + 1$;

$A(m, 0) = A(m\text{-}1, 1)$; при $m > 0$;

$A(m, n) = A(m\text{-}1, A(m, n\text{-}1))$; при $m > 0$ и $n > 0$.

6. Найти методом деления отрезка пополам минимум функции $f(x) = 7\sin^2(x)$ на отрезке [2, 6] с заданной точностью ε (например 0.01).

7. Вычислить значение $x = \sqrt{a}$, используя рекуррентную формулу $x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}}\right)$, в качестве начального значения использовать $x_0 = 0{,}5(1 + a)$.

8. Найти максимальный элемент в массиве a_i (i=1, …, n), используя очевидное соотношение $max(a_1, \ldots, a_n) = max[max(a_1, \ldots, a_{n\text{-}1}), a_n]$.

9. Вычислить значение $y(n) = \sqrt{1+\sqrt{2+\ldots+\sqrt{n}}}$.

10. Найти максимальный элемент в массиве a_i (i=1, …, n), используя соотношение (деления пополам) $max(a_1,\ldots, a_n) = max[max(a_1,\ldots, a_{n/2}), max(a_{n/2+1}, \ldots, a_n)]$.

11. Вычислить значение $y(n) = \cfrac{1}{n+\cfrac{1}{(n-1)+\cfrac{1}{(n-2)+\cfrac{1}{\cdots}}}}$.

$$\cdots+\cfrac{1}{1+\cfrac{1}{2}}$$

12. Вычислить произведение четного количества n ($n \geq 2$) сомножителей следующего вида $y = \left(\frac{2}{1}\cdot\frac{2}{3}\right)\cdot\left(\frac{4}{3}\cdot\frac{4}{5}\right)\cdot\left(\frac{6}{5}\cdot\frac{6}{7}\right)\ldots$.

13. Вычислить $y = x^n$ по следующему правилу: $y = (x^{n/2})^2$, если n четное и $y = x \cdot y^{n-1}$, если n нечетное.

14. Вычислить значение $C_n^k = \frac{n!}{k!(n-k)!}$ (значение 0! = 1).

15. Вычислить $y(n) = \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\dots}}}}$, n задает число ступеней.