

Лекция 5. Обработка массивов в языке Си++

Массив — структурированный тип данных, состоящий из фиксированного числа элементов одного типа. На рис. 5.1 представлен массив вещественных чисел X.

12.1	0.13	-1,5	0	21.9	-3.7	5.0	121.7
0-й элемент массива	1-й элемент массива	2-й элемент массива	3-й элемент массива	4-й элемент массива	5-й элемент массива	6-й элемент массива	7-й элемент массива

Рисунок 5.1: Пример массива

Одномерный массив описывают так:

тип имя_переменной [n];

где n — количество элементов в массиве, причем нумерация начинается с нуля: от 0 до n - 1.

Например:

```
int x[10];
float g[25];
```

Обратиться к элементу одномерного массива можно, указав имя массива и номер элемента в квадратных скобках.

Например,

-1	3	2	0	-8	5	1	-2
0	1	2	3	4	5	6	7

x[0] x[4]

Двумерный массив (матрицу) можно объявить так:

тип имя_переменной [n][m];

где n — количество строк (от 0 до n-1), m — количество столбцов (от 0 до m-1).

Например, `double m[3][4];`

Обращаются к элементу матрицы, указывая последовательно в квадратных скобках соответствующие индексы:

Например, `a[1][2]` — элемент матрицы `a`, находящийся в первой строке и втором столбце. Массиву, как и простой переменной, можно присвоить начальные значения в момент его описания.

Например,

```
float a[5]={1.2, (float)3/4, 5./6, 6.1, 7.8};
```

5.1. Ввод элементов массива

Блок-схема ввода алгоритма элементов массива представлена на рис. 5.2. Реализация алгоритма на C++ представлена ниже.

```
//Первая версия программы ввода элементов массива.
```

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

```
#include <math.h>
```

```
int main()
```

```
{
```

```
float x[10];
```

```
int i, n;
```

```
printf("\n N=");
```

```
scanf("%d", &n);
```

```
printf("\n Введите массив X \n");
for(i=0;i<n;i++)
scanf("%f",&x[i]);
}
```

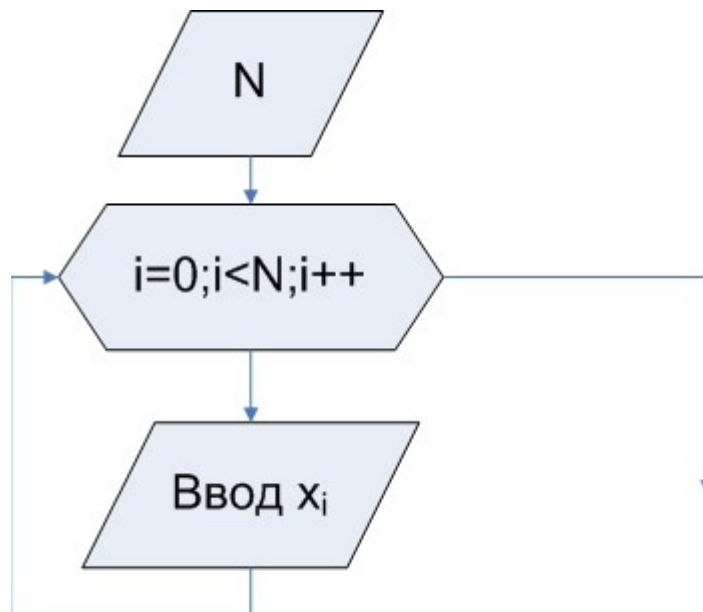


Рисунок 5.2: Блок-схема ввода массива

//Вторая версия программы ввода элементов массива.

```
#include <stdio.h>
#include <math.h>
int main()
{
float x[10],b;
int i,n;
printf("\n N=");
scanf("%d",&n);
printf("\n Введите массив X \n");
for(i=0;i<n;i++)
{
scanf("%f",&b);
x[i]=b;
} }
```

//Третья версия программы ввода элементов массива.

```
float x[10];
int i,n;
cout<<"\n N=";
cin>>n;
cout<<"\n Vvedite massiv X \n";
for(i=0;i<n;i++)
cin>>x[i];
```

5.2. Вывод элементов массива

Блок-схема вывода алгоритма элементов массива представлена на рис. 5.3.

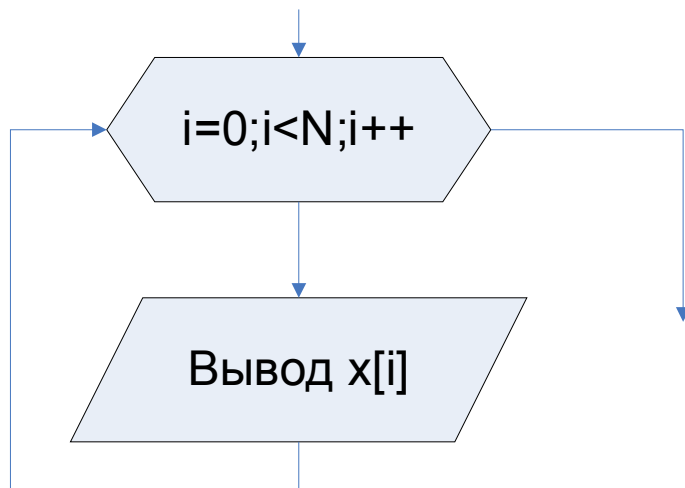


Рисунок 5.3: Блок-схема вывода элементов массива

Реализация алгоритма на С++ представлена ниже. При организации вывода элементов массива можно использовать специальные символы `\t \n`.

```

printf("\n Массив X\n");
for (i=0; i<n; i++)
printf("%g\t", x[i]);
printf("\n");
cout<<"\n Massiv X \n";
    for (i=0; i<n; i++)
        cout<<x[i]<<"\t";
  
```

5.3. Основные алгоритмы обработки массивов

5.3.1 Алгоритм вычисления суммы элементов массива

Дан массив X , состоящий из n элементов. Найти сумму элементов этого массива. Процесс накапливания суммы элементов массива и практически ничем не отличается от суммирования значений некоторой числовой последовательности. Переменной S присваивается значение равное нулю, затем последовательно суммируются элементы массива X . Блок-схема алгоритма расчета суммы приведена на рис. 5.4.

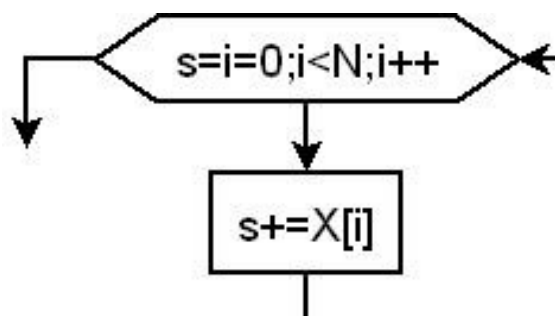


Рисунок 5.4: Блок-схема нахождения суммы элементов массива

Реализация на С++.

```

for (s=i=0; i<N; i++)
s+=X[i];
//Это можно записать и так
//for (s=i=0; i<N; s+=X[i], i++);
  
```

5.3.2 Алгоритм вычисления произведения элементов массива

Дан массив X , состоящий из n элементов. Найти произведение элементов этого массива. Решение этой задачи сводится к тому, что значение переменной P , в которую предварительно была записана единица, последовательно умножается на значение i го элемента массива. Блок-схема алгоритма приведена на рис. 5.5.

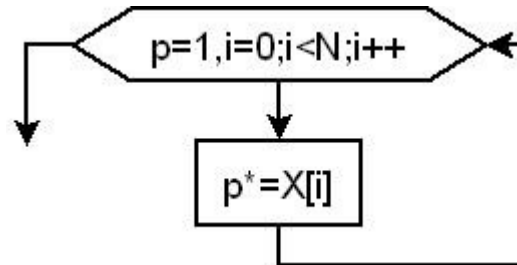


Рисунок 5.5: Алгоритм нахождения произведения элементов массива

Реализация на C++.

```
for (P=1, i=0; i<n; i++)  
P*=X[i];
```

5.3.3. Поиск максимального элемента и его номера

Алгоритм решения задачи следующий. Пусть в переменной с именем max хранится значение максимального элемента массива, а в переменной с именем $nmax$ его номер. Предположим, что нулевой элемент массива является максимальным и запишем его в переменную max , а в $nmax$ его номер (то есть ноль). Затем все элементы, начиная с первого, сравниваем в цикле с максимальным. Если текущий элемент массива оказывается больше максимального, то записываем его в переменную max , а в переменную $nmax$ текущее значение индекса i . Процесс определения максимального элемента в массиве изображен при помощи блок-схемы на рис. 5.6.

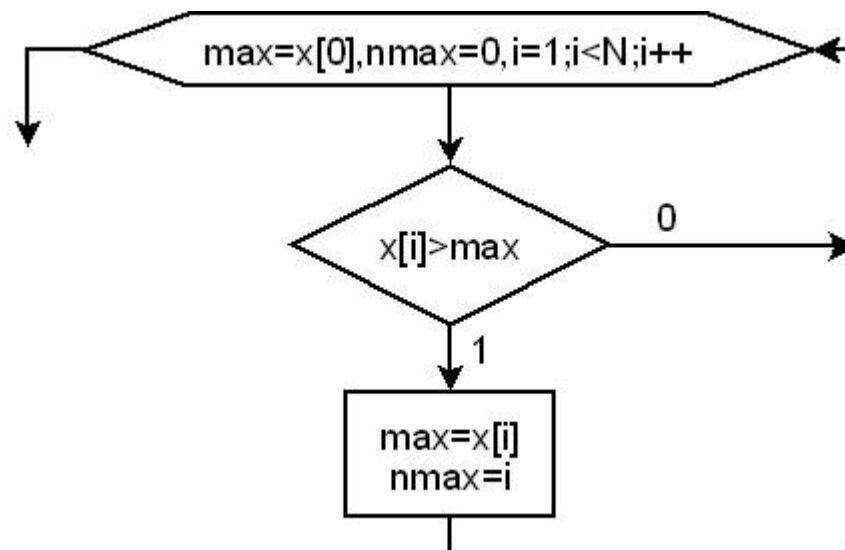


Рисунок 5.6: Поиск максимального элемента массива и его номера

Реализация алгоритма в C++.

```
for (max=X[0], nmax=0, i=1; i<n; i++)  
if (x[i]>max)  
{  
max=x[i]; nmax=i;
```

```
}
```

Алгоритм поиска минимума будет отличаться знаком в блоке сравнения. Значительно более интересной является задача поиска минимального (максимального) элемента массива, среди элементов массива, удовлетворяющих некоторому условию. Рассмотрим на примере поиска минимального значения, среди положительных элементов массива.

```
for (i=k=0; i<n; i++)  
// Если элемент положительный,  
if (x[i]>0)  
{  
// то увеличиваем количество положительных элементов на 1.  
k++;  
// Если это первый положительный элемент, то объявляем его  
// минимальным, иначе  
if (k==1) {min=x[i]; nmin=i;}  
// сравниваем его с минимальным  
else if (x[i]<min)  
{  
min=x[i]; nmin=i;  
}  
}
```

5.3.4. Алгоритм удаления элемента из массива

Пусть необходимо удалить из массива, состоящего из семи элементов, четвертый по номеру элемент. Для этого необходимо выполнить смещение элементов.

```
x[3]=x[4];x[4]=x[5];x[5]=x[6];
```

Блок-схема этого процесса представлена на рис. 5.7.

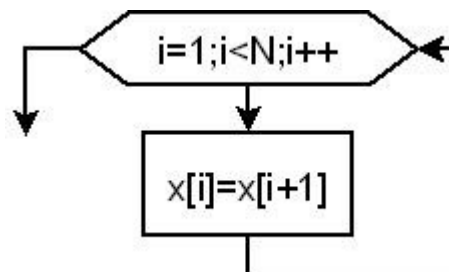


Рисунок 5.7: Удаление четвертого по счету элемента

Блок-схема удаления элемента с номером М из массива X, в котором N элементов изображена на рис. 5.8.

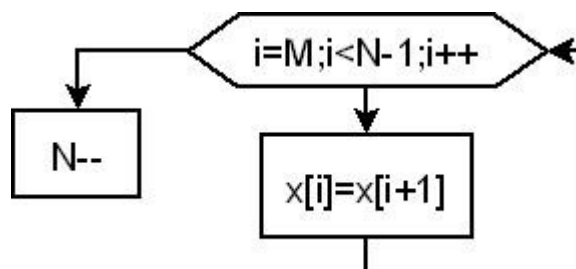


Рисунок 5.8: Блок-схема удаления элемента из массивов

Реализация в C++.

```
for (i=M; i<N-1; i++)  
x[i]=x[i+1];  
N--
```

После удаления следует учитывать, что изменилась нумерация (все номера уменьшились на 1) элементов, начиная с номера М, поэтому если удалять несколько элементов подряд не надо переходить к следующему.

ЗАДАЧА 5.1. Удалить элементы с 4-го по 8-й в массиве из N элементов. Блок-схема представлена на рис. 5.9.

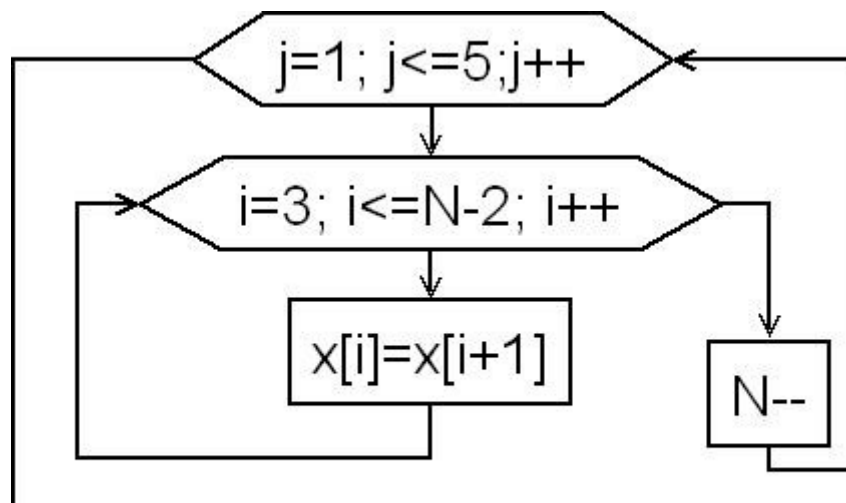


Рисунок 5.9: Блок-схема решения задачи 5.1

Реализация блок-схемы в C++.

```
for (j=1; j<=5; j++, N--)
for (i=3; i<=N-2; i++) X[i]=X[i+1];
```

Программа решения задачи 5.1 приведена ниже.

```
int main()
{
    floxt x[20];
    int i, j, n;
    cout<<"n=";
    cin>>n;
    cout<<"Massiv x\n";
    for (i=0; i<n; i++)
        cin>>x[i];
    for (j=1; j<=5; j++)
    {
        for (i=3; i<=n-2; i++)
            x[i]=x[i+1];
        n--;
    }
    cout<<"Massiv x\n";
    for (i=0; i<n; i++)
        cout<<"x ("<<i<<" )="<<x[i]<<"\t";
    cout<<endl;
    return 0;
}
```

5.3.5. Упорядочение элементов массива

Алгоритм *сортировки выбором по возрастанию*¹ приведен в виде блок-схемы на рис. 5.10. Идея алгоритма заключается в следующем. В массиве состоящем из n элементов ищем самый большой элемент и меняем его местами с последним элементом. Повторяем алгоритм поиска максимального элемента, но последний (n-1)-й элемент не рассматривается, так как он уже

¹ Для сортировки по убыванию надо поменять в блок-схеме знак «>» на знак «<».

занял свою позицию. Найденный максимальный элемент меняем местами с (n-2)-м элементом. Описанную выше операцию поиска проводим n-1 раз, до полного упорядочивания элементов в массиве.

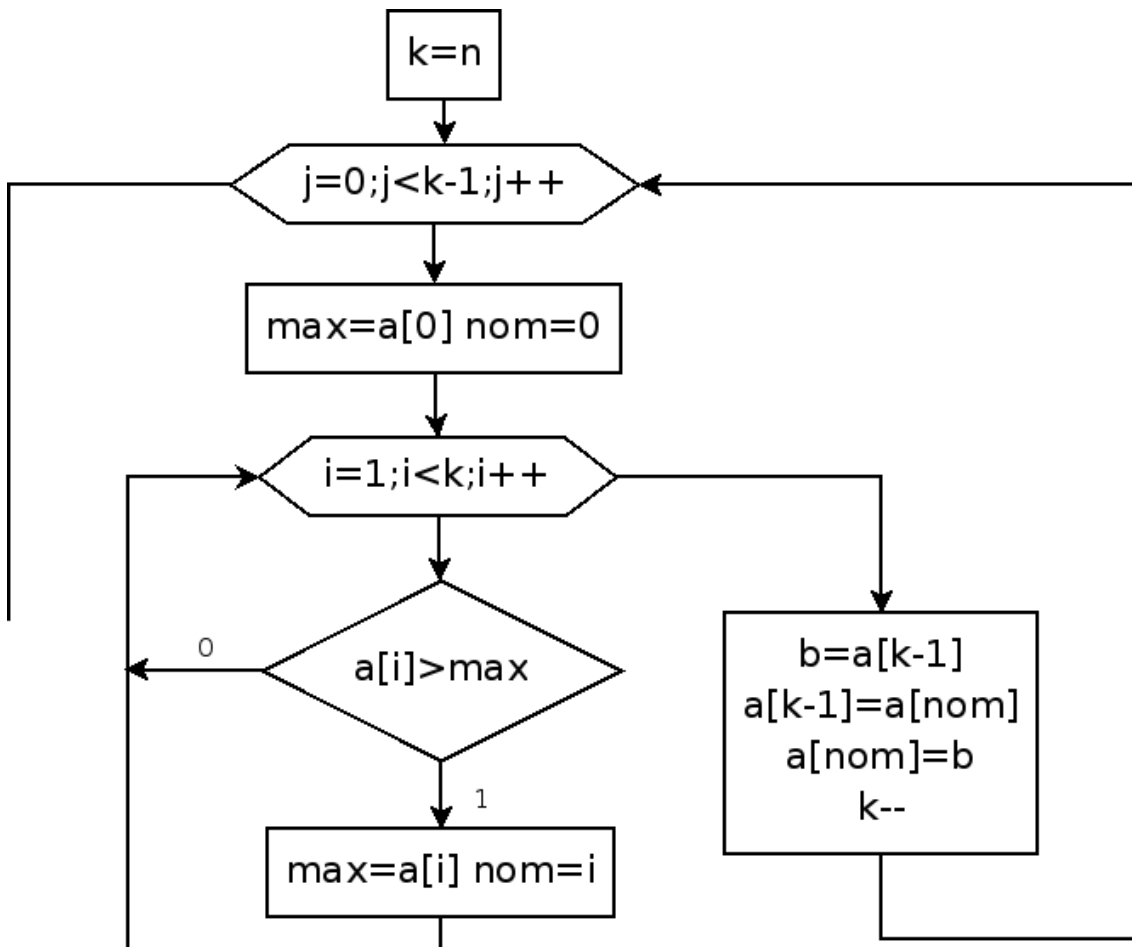


Рисунок 5.10: Алгоритм сортировки выбором
Реализация алгоритма упорядочивания на C++

```

int main()
{
    float b,max,a[20];
    int i,j,n,k,nom;
    cout<<"n=";
    cin>>n;
    cout<<"Massiv a\n";
    for(i=0;i<n;i++)
        cin>>a[i];
    k=n;
    for(j=0;j<=k-2;j++)
    {
        max=a[0];nom=0;
        for(i=1;i<k;i++)
            if(a[i]>max)
            {
                max=a[i];
                nom=i;
            }
        b=a[k-1];
        a[k-1]=a[nom];
        a[nom]=b;
    }
}
  
```

```

        k--;
    }
    cout<<"Massiv a\n";
    for(i=0;i<n;i++)
        cout<<"a ("<<i<<" )="<<a[i]<<"\t";
    cout<<endl;
    return 0;
}

```

Сортировка *методом пузырька*. Сортировка пузырьковым методом является наиболее известной. Ее популярность объясняется запоминающимся названием, которое происходит из-за подобия процессу движения пузырьков в резервуаре с водой, когда каждый пузырек находит свой собственный уровень, и простотой алгоритма. Сортировка методом «пузырька» использует метод обменной сортировки и основана на выполнении в цикле операций сравнения и при необходимости обмена соседних элементов. Рассмотрим алгоритм пузырьковой сортировки более подробно.

Сравним нулевой элемент массива с первым, если нулевой окажется больше первого, то поменяем их местами. Те же действия выполним для первого и второго, второго и третьего, i го и $(i+1)$ го предпоследнего и последнего элементов. В результате этих действий самый большой элемент станет на последнее $(n-1)$ -е место. Теперь повторим данный алгоритм сначала, но последний $(n-1)$ -й элемент, рассматривать не будем, так как он уже занял свое место. После проведения данной операции самый большой элемент оставшегося массива станет на $(n-2)$ е место. Так повторяем до тех пор, пока не упорядочим весь массив. Блок-схема сортировки элементов массива по возрастанию² представлена на рис. 5.11.

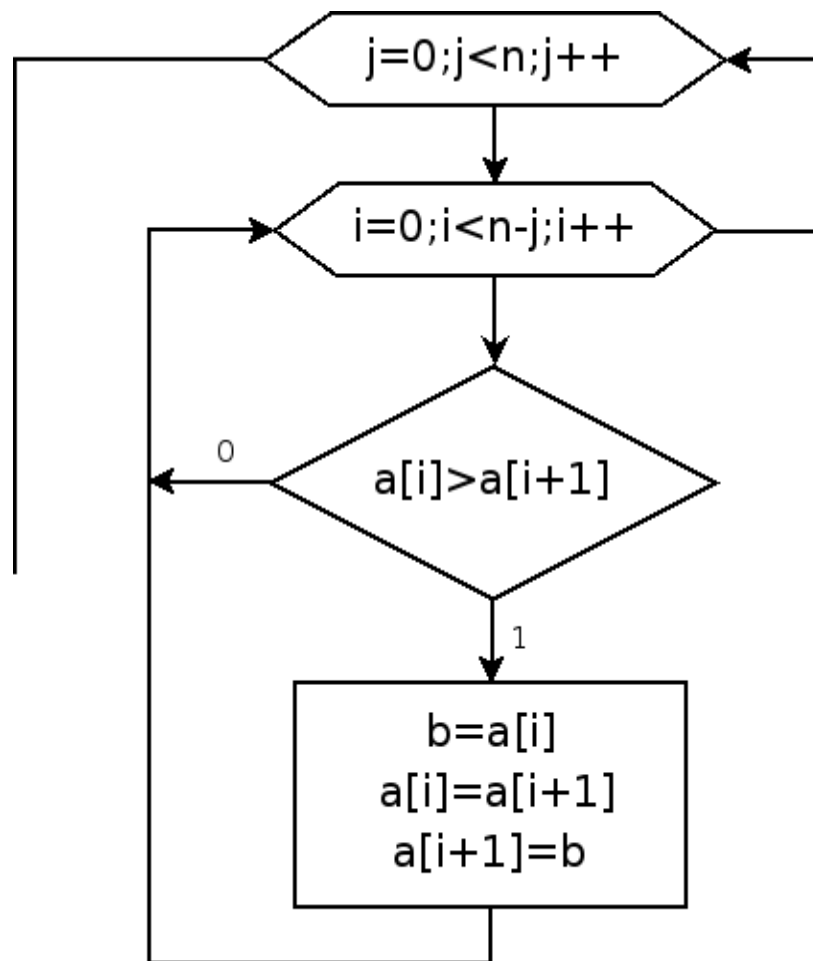


Рисунок 5.11: Упорядочивание массива по возрастанию методом пузырька

2 Для сортировки по убыванию надо поменять в блок-схеме знак «>» на знак «<».


```

int main()
{
    float b,max,a[20];
    int i,j,n,k,nom;
    cout<<"n=";
    cin>>n;
    cout<<"Massiv a\n";
    for(i=0;i<n;i++)
        cin>>a[i];
    for(j=1;j<=n-1;j++)
        for(i=0;i<=n-1-j;i++)
            if (a[i]>a[i+1])
            {
                b=a[i];
                a[i]=a[i+1];
                a[i+1]=b;
            }
    cout<<"Massiv a\n";
    for(i=0;i<n;i++)
        cout<<"a ("<<i<<" )="<<a[i]<<"\t";
    cout<<endl;
    return 0;
}

```

5.3.6. Запись положительных элементов массива A в массив B

Блок-схема решения этой задачи представлена на рис. 5.12

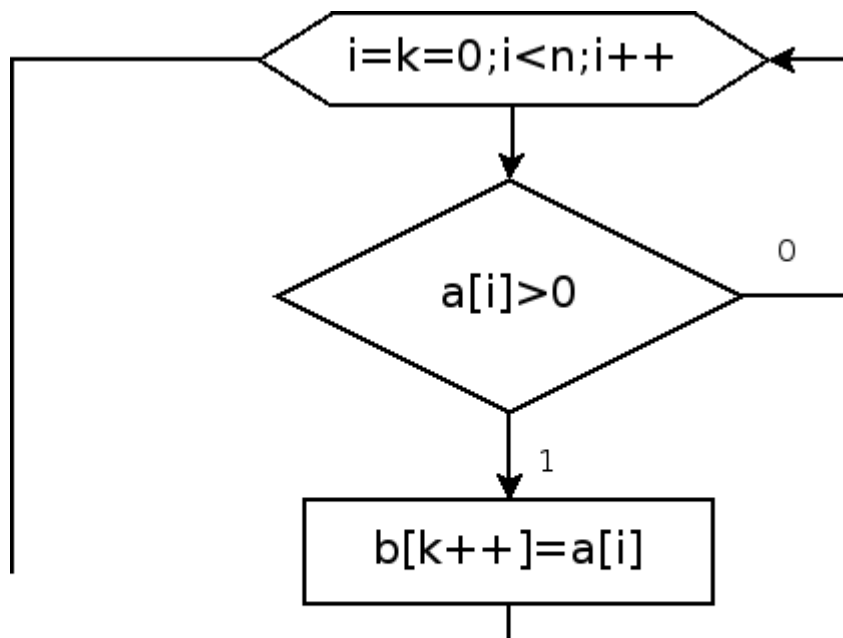


Рисунок 5.12: Алгоритм перезаписи из одного массива в другой

Реализация алгоритма на C++.

```

int main()
{
    float a[20],b[20];
    int i,n,k;
    cout<<"n=";    cin>>n;
    cout<<"Massiv a\n";

```

```

for (i=0; i<n; i++)
    cin >> a[i];
for (k=i=0; i<n; i++)
    if (a[i]>0)    b[k++] = a[i];
cout << "Massiv b\n";
for (i=0; i<k; i++)    cout << "b (" << i << ") = " << b[i] << "\t";
cout << endl;
return 0; }

```

5.3.7. Вставка

Пусть массив $X(N)$ упорядочен по возрастанию, необходимо в него вставить элемент b , не нарушив упорядоченности массива. Блок-схема решения задачи представлена на рис.5.13.

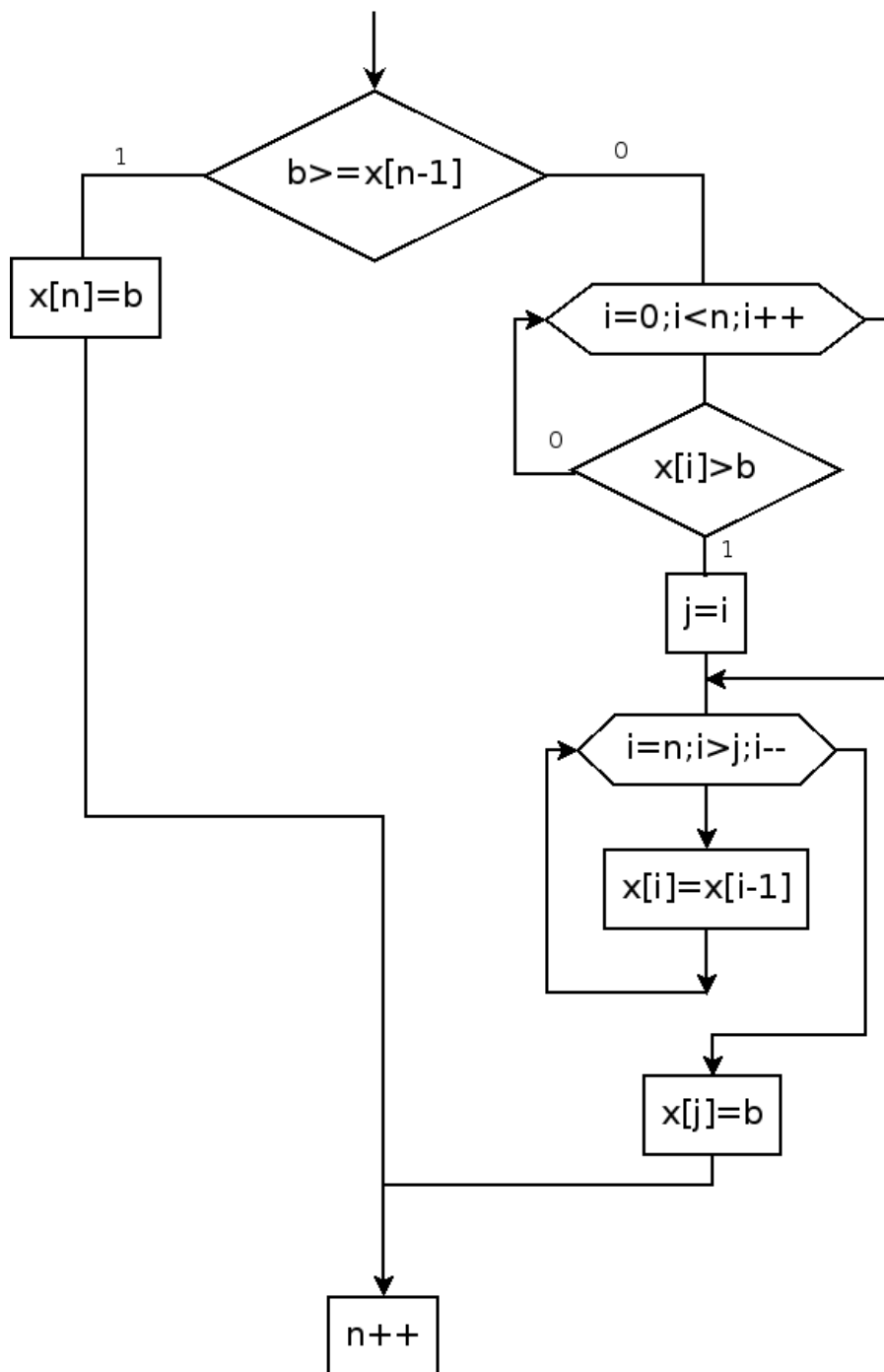


Рисунок 5.13: Вставка элемента в упорядоченный массив

```
int main()
```

```

{   float x[20],b;
    int i,j,n;
    cout<<"n=";
    cin>>n;
    cout<<"Massiv x\n";
    for (i=0;i<n;i++)
        cin>>x[i];
    cout<<"b=";    cin>>b;
    if (b>=x[n-1])
        x[n]=b;
    else
    {   for (i=0;i<n;i++)
        if (x[i]>b)
            {j=i;break;}
        for (i=n;i>j;i--)
            x[i]=x[i-1];
        x[j]=b;
    }
    n++;
    cout<<"Massiv x\n";
    for (i=0;i<n;i++)
cout<<"x ("<<i<<")="<<
x[i]<<"\t";
    cout<<endl;
    return 0;}

```

5.4. Указатели, динамические массивы

Как видно из рассмотренных ранее примеров, в Си++ массивы статические, их размер задается при описании. Это не всегда удобно, кроме того, при решении некоторых задач заранее неизвестен размер формируемого массива.

В Си++ существуют динамические массивы — массивы переменной длины, они определяются с помощью указателей.

5.4.1. Указатели в C++

Указатель — переменная, значением которой является адрес памяти, по которому хранится объект определенного типа. При объявлении указателей всегда указывается тип объекта, который будет храниться по данному адресу.

Указатель описывается следующим образом:

type * name;

Здесь name — переменная, объявляемая, как указатель. По этому адресу (указателю) храниться значение типа type.

Например:

int *i;

Объявляем указатель (адрес) i. По этому адресу будет храниться переменная типа int. Переменная i указывает на тип данных int.

float *x,*z;

Объявляем указатели с именами x и z, которые указывают на переменные типа float.

5.4.2. Операции * и & при работе с указателями

При работе с указателями в основном используются операции & и *. Операция & возвращает адрес своего операнда.

Например, если объявлена переменная `a` следующим образом:

```
float a;
```

то оператор

```
adr_a=&a;
```

записывает в переменную `adr_a` адрес переменной `a`, переменная `adr_a` должна быть указателем на тип `float`. Ее следует описать следующим образом:

```
float *adr_a;
```

Операция `*` выполняет действие, обратное операции `&`. Она возвращает значение переменной, хранящееся по заданному адресу.

Например, оператор

```
a=*adr_a;
```

записывает в переменную `a` вещественное значение, хранящееся по адресу `adr_a`.

5.4.3. Операция присваивания указателей

Значение одного указателя можно присвоить другому. Если указатели одного типа, то один можно присваивать другому с помощью обычной операции присваивания.

Рассмотрим следующий пример

```
#include <stdio.h>
#include <math.h>
int main()
{
float PI=3.14159, *p1, *p2;
p1=p2=&PI;
printf("По адресу p1=%p хранится *p1=%g\n", p1, *p1);
printf("По адресу p2=%p хранится *p2=%g\n", p2, *p2);
}
```

В этой программе определены: вещественная переменная `PI=3.14159` и два указателя на тип `float` `p1` и `p2`. Затем в указатели `p1` и `p2` записывается адрес переменной `PI`. Операторы `printf` выводят на экран адреса `p1` и `p2` и значения, хранящиеся по этим адресам. Для вывода адреса используется спецификатор типа `%p`. В результате работы этой программы в переменных `p1` и `p2` будет храниться значение одного и того же адреса, по которому хранится вещественная переменная `PI=3.14159`.

По адресу `p1=0012FF7C` хранится `*p1=3.14159`

По адресу `p2=0012FF7C` хранится `*p2=3.14159`

Если указатели ссылаются на различные типы, то при присваивании значения одного указателя другому, необходимо использовать преобразование типов. Без преобразования можно присваивать любому указателю указатель `void *`. Рассмотрим пример работы с указателями различных типов.

```
#include <stdio.h>
#include <math.h>
int main()
{
float PI=3.14159, *p1;
double *p2;
//В переменную p1 записываем адрес PI
p1=&PI;
//указателю на double присваиваем значение, которое ссылается на
//тип float.
p2=(double *)p1;
```

```
printf("По адресу p1=%p хранится *p1=%g\n", p1, *p1);
printf("По адресу p2=%p хранится *p2=%e\n", p2, *p2);
}
```

По адресу p1=0012FF7C хранится *p1=3.14159

По адресу p2=0012FF7C хранится *p2=2.642140e-308

В указателях p1 и p2 хранится один и тот же адрес, но значения, на которые они ссылаются, оказываются разными. Это связано с тем, указатель типа *float адресует 4 байта, а указатель *double 8 байт. После присваивания p2=(double *)p1; при обращении к *p2 происходит следующее: к переменной, хранящейся по адресу p1, дописывается еще 4 байта из памяти. В результате значение *p2 не совпадает со значением *p1.

А что произойдет в результате следующей программы?

```
#include <stdio.h>
#include <math.h>
int main()
{
double PI=3.14159, *p1;
float *p2;
p1=&PI;
p2=(float *)p1;
printf("По адресу p1=%p хранится *p1=%g\n", p1, *p1);
printf("По адресу p2=%p хранится *p2=%e\n", p2, *p2);
}
```

После присваивания p2=(double *)p1; при обращении к *p2 происходит следующее: из переменной, хранящейся по адресу p1, выделяется только 4 байта. В результате и в этом случае значение *p2 не совпадает со значением *p1.

ВЫВОД. При преобразовании указателей разного типа приведение типов разрешает только синтаксическую проблему присваивания. Следует помнить, что операция * над указателями различного типа, ссылающимися на один и тот же адрес, возвращает различные значения.

Если есть следующий оператор

float *p;

то в переменной p хранится адрес, а с помощью конструкции *p можно получить значение, хранящееся по адресу p.

В случае использования оператора

float p;

то для вычисления адреса используется конструкция &p, а в переменной p находится вещественное значение.

5.4.4. Арифметические операции над адресами

Над адресами в языке Си определены следующие операции:

- суммирование, можно добавлять к указателю целое значение;
- вычитание, можно вычитать указатели или вычитать из указателя целое число.

Однако при выполнении арифметических операций есть некоторые особенности. Рассмотрим их на следующем примере.

```
double *p1;
float *p2;
int *i;
```

```
p1++
```

```
p2++;  
i++;  
}
```

Операция `p1++` увеличивает значение адреса на 8, операция `p2++` увеличивает значение адреса на 4, а операция `i++` на 2. Операции адресной арифметики выполняются следующим образом:

- операция увеличения приводит к тому, что указатель будет сдвигаться на следующий объект базового типа (для `p1` это `double`, для `p2` `float` для `i` `int`);
- операция уменьшения приводит к тому, что указатель, ссылается на предыдущий объект базового типа.
- после операции `p1=p1+n`, указатель будет передвинут на `n` объектов базового типа; `p1+n` как бы адресует `n`-й элемент массива, если `p1` адрес начала массива .

5.4.5. Использование адресов и указателей при работе с массивами. Динамические массивы.

С помощью указателей в Си можно выделить участок памяти (динамический массив) заданного размера для хранения данных определенного типа. Для этого необходимо выполнить следующие действия :

1. Описать указатель (например, переменную `p`) определенного типа.
2. Начиная с адреса, определенного указателем, с помощью функций `calloc`, `malloc` или операции `new` выделить участок памяти определенного размера. После этого `p` будет адресом первого элемента выделенного участка оперативной памяти (0-й элемент массива), `p+1` будет адресовать следующий элемент в выделенном участке памяти (1-й элемент динамического массива), `& p+i` является адресом `i`-го элемента. Необходимо только следить, чтобы не выйти за границы выделенного участка памяти. К `i`-му элементу динамического массива `p` можно обратиться одним из двух способов `*(p+i)` или `p[i]`.
3. Когда участок памяти будет не нужен, его можно освободить с помощью функции `free()`, операции `delete`.

Перед подробным описанием работы с динамическими переменными, рассмотрим функции `calloc`, `malloc`, `realloc` и `free` и операции `new` и `delete`.

Единственным параметром функции `malloc` является целое беззнаковое значение, определяющее размер выделяемого участка памяти в байтах. Функция `malloc` возвращает бестиповый указатель (`void *`) на выделенный участок памяти. Обращение к функции `malloc` имеет вид

```
void *malloc(n);
```

здесь `n` определяет размер выделяемого участка памяти в байтах, функция вернет значение `NULL`, если выделить память не удалось и указатель на выделенный участок памяти, при успешном выделении.

В качестве примера использования функции `malloc` решим следующую задачу.

Найти сумму элементов динамического массива вещественных чисел.

```
#include <iostream>  
#include <malloc.h>  
using namespace std;  
int main()  
{  
    int i,n;  
    float *a,s;
```

```

    cout<<"n=";
    cin>>n;
// Выделение памяти для динамического массива a.
a=(float *)malloc(n*sizeof(float));
    cout << "Vvedite massiv A";
    for(i=0;i<n;i++)
        // cin>>a[i];
        cin>>*(a+i);
    for(s=0,i=0;i<n;i++)
        //s+=a[i];
        s+=*(a+i);
    cout << "S="<<s;
//Освобождение памяти.
    free(a);
    return 0;
}

```

Кроме функции malloc для выделения памяти в Си есть функция calloc. Ее особенностью является обнуление всех выделенных элементов. Обращение к функции имеет вид:

void *calloc (num, size);

Функция calloc выделяет num элементов по size байт и возвращает указатель на выделенный участок или NULL при невозможности выделить память.

Ниже приведено решение задачи поиска суммы элементов динамического массива вещественных чисел с помощью функции calloc.

```

#include <iostream>
#include <malloc.h>
using namespace std;
int main()
{
    int i,n;
    float *a,s;
    cout<<"n=";
    cin>>n;
// Выделение памяти для динамического массива a.
a=(float *)calloc(n,sizeof(float));
    cout << "Vvedite massiv A";
    for(i=0;i<n;i++)
        cin>>*(a+i);
    for(s=0,i=0;i<n;i++)
        s+=*(a+i);
    cout << "S="<<s;
//Освобождение памяти.
    free(a);
    return 0;
}

```

Функция realloc изменяет размер выделенной ранее памяти, обращение к ней имеет вид:

char *realloc(void *p, size);

Функция изменяет размер участка памяти, на который указывает p, новый размер участка памяти size. Если при этом пришлось изменить месторасположение участка памяти, то новое его месторасположение и возвращается в качестве результата. Если в качестве p передается NULL, то функция realloc работает аналогично функции malloc.

Для освобождения выделенной памяти используется функция free. Обращение к ней имеет вид

void free(void *p);

Освобождает память, выделенную память, p указатель на участок память, ранее выделенный функциями calloc, malloc или realloc.

Для выделения памяти в C++ есть еще операция new.

Ниже приведено решение задачи поиска суммы элементов динамического массива вещественных чисел с помощью операции new.

```
#include <iostream>
using namespace std;
int n;
float *a;
cin>>n;
a=new float[n];
int main()
{
    int i,n;
    float *a,s;
    cout<<"n=";
    cin>>n;
    // Выделение памяти для динамического массива a.
    a=new float[n];
    cout << "Vvedite massiv A";
    for(i=0;i<n;i++)
        // cin>>a[i];
        cin>>*(a+i);
    for(s=0,i=0;i<n;i++)
        //s+=a[i];
        s+=*(a+i);
    cout << "S="<<s;
    //Освобождение памяти.
    delete [] a;
    return 0;
}
```

Указатели можно использовать в качестве аргументов функций. Так, если в роли параметра функции выступает массив, то в функцию передается указатель на его первый элемент. C++ автоматически передает массив в функцию, используя его адрес. В результате вызываемые функции могут изменять значения элементов в исходных массивах и возвращать их в главную функцию. Информация о количестве элементов массива должна передаваться через отдельный параметр.

5.5. Примеры программ

ЗАДАЧА 5.2. В заданном массиве найти длину самой длинной серии элементов, состоящей из единиц.

Блок-схема задачи представлена на рис. 5.14. Переменная Fl принимает значение 1, если была серия, состоящая элементов из единиц, 0 если такой серии не была. В переменной k хранится длина текущей серии, в переменной max количество элементов в самой длинной серии.

```
int main()
{
    int *x,max,i,k,fl,n,b;
    cout<<"\n n=";
    cin>>n;
```



```
x=new int[n];
```

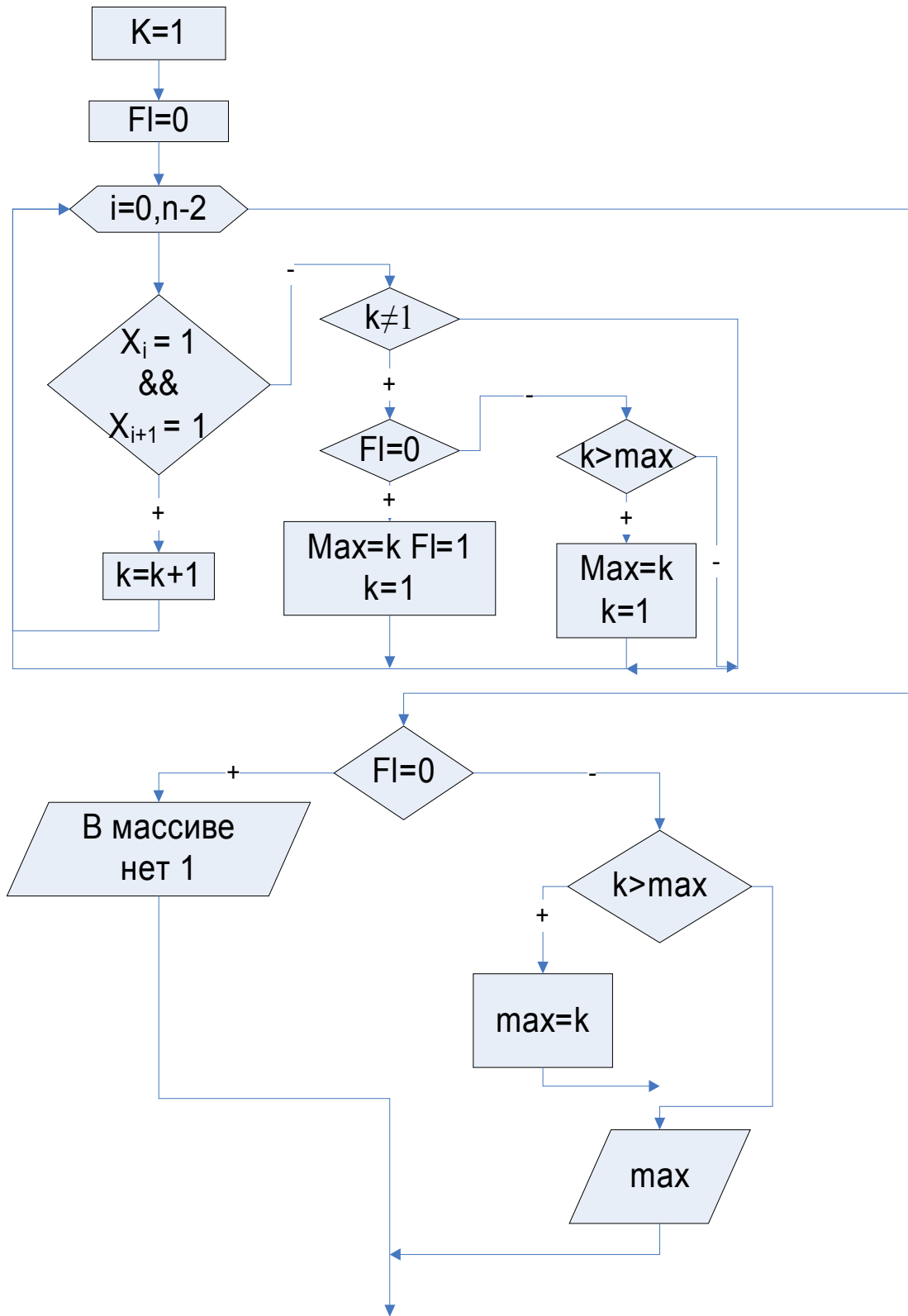


Рисунок 5.14: Блок-схема решения задачи 5.2

```

for (i=0;i<n;i++)
{
cout<<"\n x("&<<i<<"")=", i);
cin>>x[i];
}

```

```

for (k=1, fl=0, i=0; i<n-1; i++)
{
if (*(x+i)==1 && *(x+i+1)==1)
k++;
else if (k!=1)
{
if (!fl)
{
max=k;
fl=1;
k=1;
}
else if (k>max)
max=k;
k=1;
}
}
if (fl==0)
cout<<"V massive net seriy iz 1\n";
else
{
if (k>max)
max=k;
cout<<endl<<max;
}
delete [] x;
return 0;
}

```

ЗАДАЧА 5.2. Из массива целых чисел удалить все простые числа меньше среднего арифметического. Полученный массив упорядочить по возрастанию.

Кроме главной функции main() при решении этой задачи необходимо написать следующие функции:

- **bool prostoe(int n)**, которая будет проверять является ли число n простым (см. рис. 5.15);
- **void udal(int *x, int m, int *n)** функция удаления элемента с номером m в массиве x из n элементов (см. рис. 5.16);
- **void upor(int *x, int N, bool pr=true)** функция упорядочивания массива по возрастанию (если pr=true) или по убыванию (если pr=false) (см. рис. 5.17);
- **float sr_arifm(int x[], int n)** функция вычисления среднего арифметического в массиве x из n элементов (см. рис. 5.18).

На рис. 5.19 приведена блок-схема решения задачи 5.3.

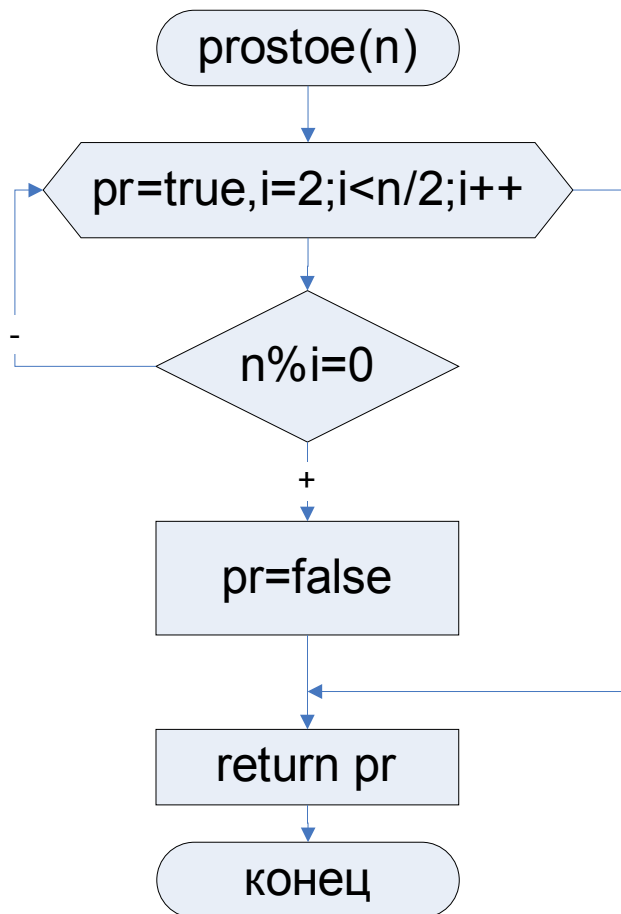


Рисунок 5.15: Блок-схема функции *prostoe*

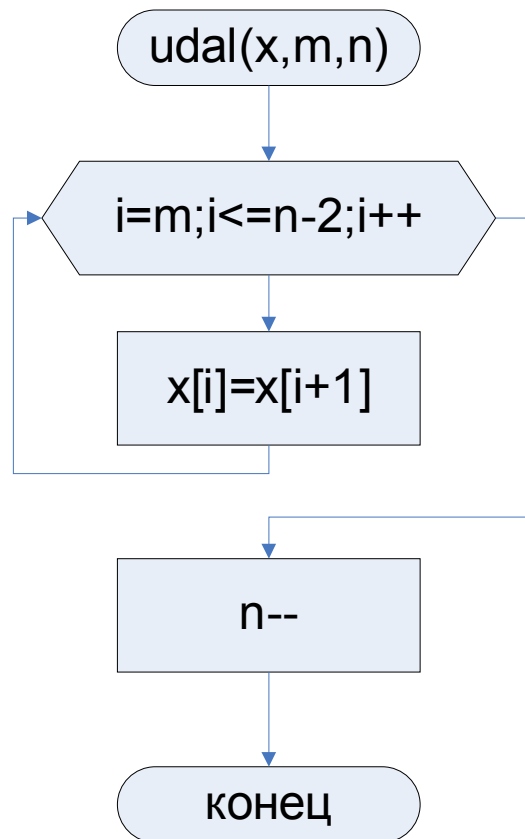


Рисунок 5.16: Блок-схема функции *udal*

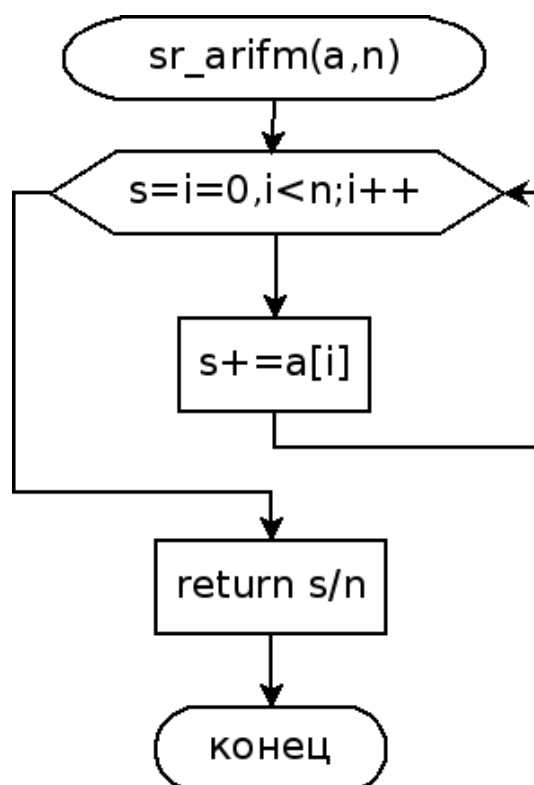


Рисунок 5.17: Блок-схема функции *sr arifm*

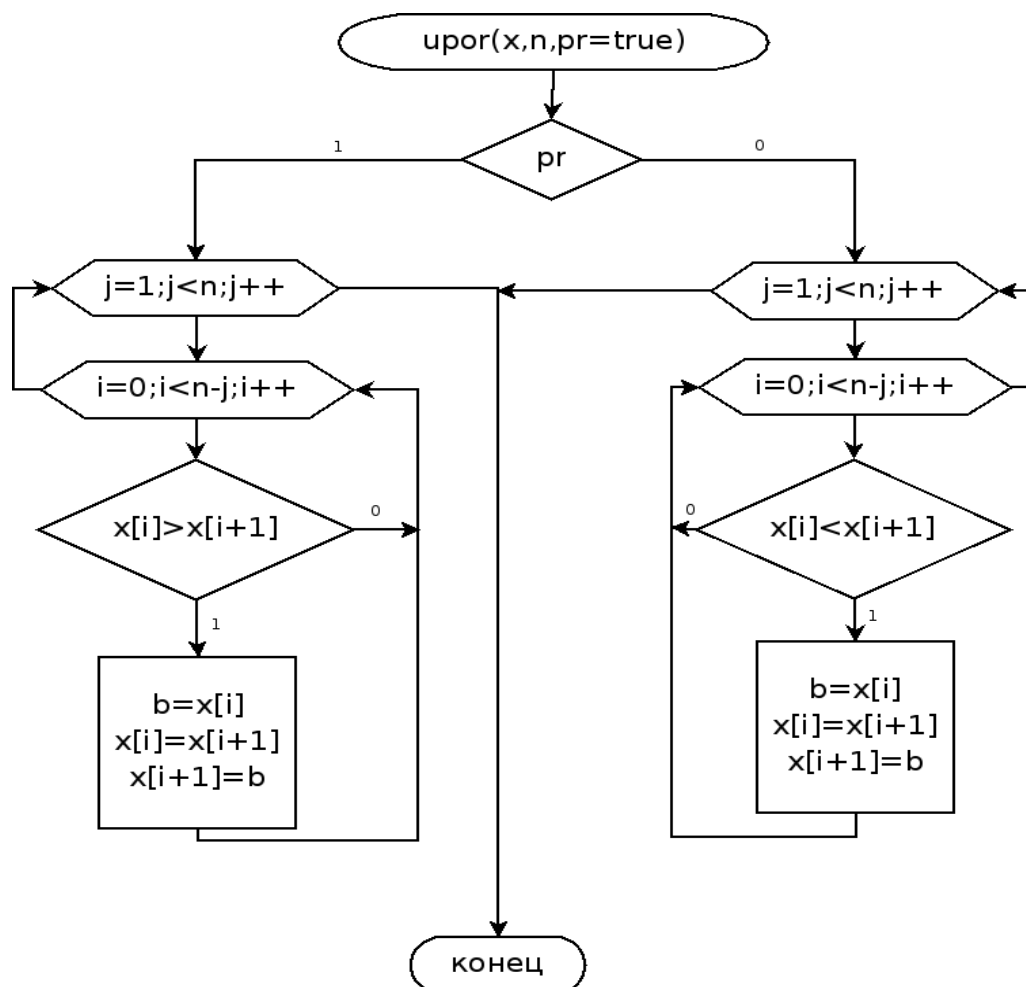


Рисунок 5.18: Блок-схема функции *упор*

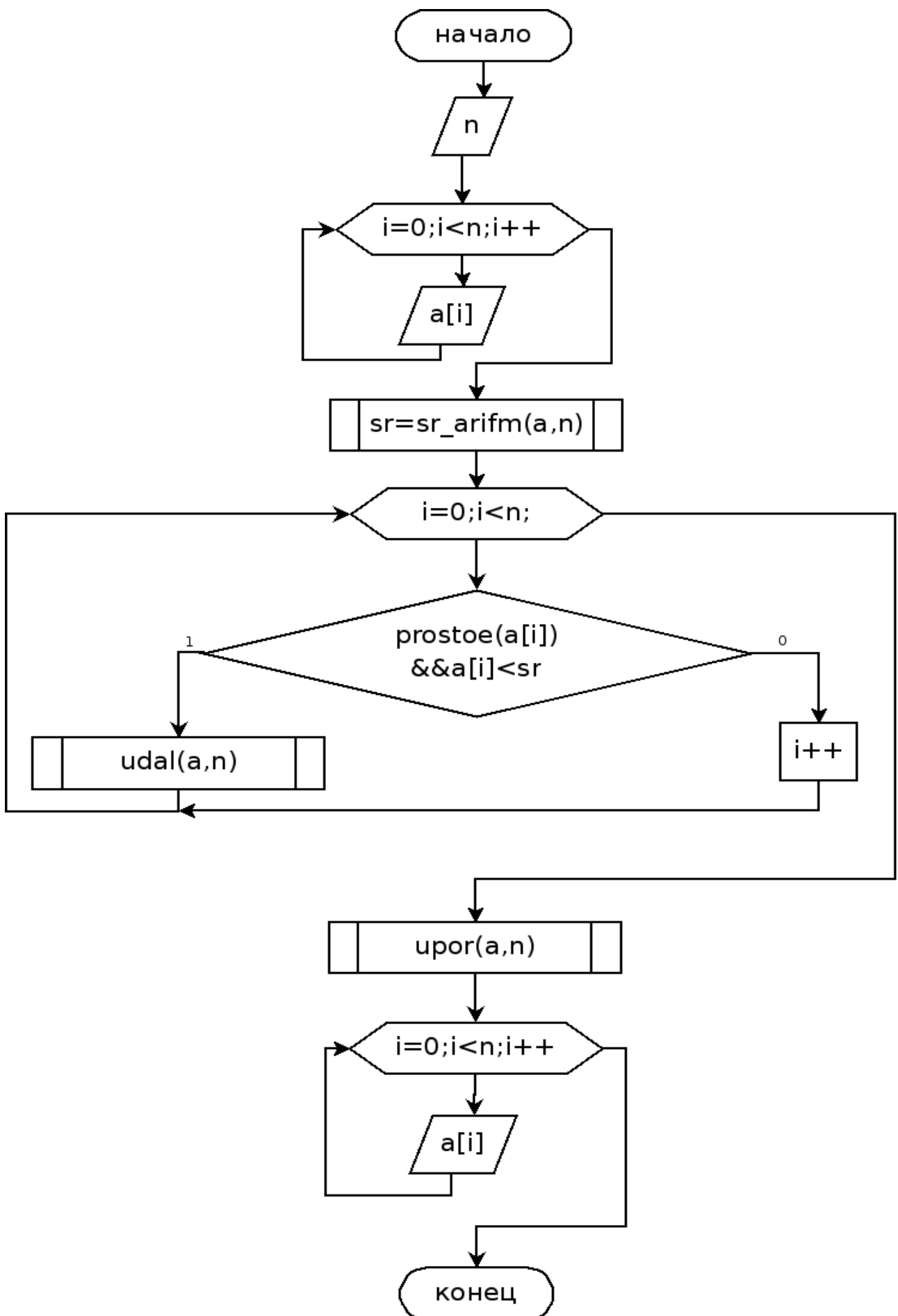


Рисунок 5.19: Блок-схема решения задачи 5.3

Ниже приведен текст решения задачи на C++.

```
#include <iostream>
#include <malloc.h>
using namespace std;
//Функция вычисления среднего значения.
float sr_arifm(int *x, int n)
{
int i; float s=0;
for(i=0;i<n;s+=x[i],i++);
if (n>0) return(s/n);
else return 0;
}
//Функция для определения простого числа:
bool prostoe(int n)
{
bool pr; int i;
for(pr=true,i=2;i<=n/2;i++)
if(n%i==0) {pr=false;break;}
return(pr);
}
//Функция удаления элемента из массива.
void udal(int *x, int m, int *n)
{
int i;
for(i=m;i<*n-1;*(x+i)=*(x+i+1),i++);
--*n;
realloc((int *)x,*n*sizeof(int));
}
//Функция сортировки массива.
void upor(int *x, int n, bool pr=true)
{
int i,j,b;
if (pr)
{
for(j=1;j<=n-1;j++)
for(i=0;i<=n-1-j;i++)
if (*(x+i)>*(x+i+1))
{
b=*(x+i);
*(x+i)=*(x+i+1);
*(x+i+1)=b;
}
}
else
for(j=1;j<=n-1;j++)
for(i=0;i<=n-1-j;i++)
if (*(x+i)<*(x+i+1))
{
b=*(x+i);
*(x+i)=*(x+i+1);
*(x+i+1)=b;
}
}
}
```

```

int main()
{
int *a,n,i; float sr;
cout<<"n="; cin>>n;          //Ввод размерности массива.
a=(int *)calloc(n,sizeof(int)); //Выделение памяти.
cout << "Vvedite massiv A\n";
for(i=0;i<n;i++) cin>>*(a+i); //Ввод массива.
sr=sr_arifm(a,n);           //Вычисление среднего
арифметического.
cout<<"sr="<<sr<<"\n";      //Вывод среднего арифметического.
for(i=0;i<n;)
{
if(prostoe(*(a+i))&& *(a+i)<sr) //Если число простое и меньше
среднего,
udal(a,i,&n);           //удалить его из массива,
else i++;              //иначе, перейти к следующему элементу.
}
cout << "Massiv A\n";      //Вывод модифицированного массива.
for(i=0;i<n;i++) cout<<*(a+i)<<"\t";
cout<<"\n";
upor(a, n);            //Сортировка массива.
cout << "Upor Massiv A\n"; //Вывод упорядоченного массива.
for(i=0;i<n;i++) cout<<*(a+i)<<"\t";
cout<<"\n";
free(a);               //Освобождение памяти.
return 0;
}

```

Результат работы программы

n=10

Vvedite massiv A

6 20 5 3 10 301 17 11 6 8

sr=38.7

Massiv A

6 20 10 301 6 8

Upor Massiv A

6 6 8 10 20 301