Лекция № 5

Разные мелочи

Именованные константы. Программа на языке С (и С++) при обработке сложных данных использует указатели и динамически выделяемую память. Но при разработке программы некоторые значения являются неизменными и нет причин полагать, что они будут меняться. Такие значения задают константами. Но иногда приходится такие константы изменять и перекомпилировать программу. Для упрощения такого процесса лучше использовать именованные константы, то есть конструкцию с использованием ключевого слова `const`. Синтаксис команды следующий:

```
const <имя> = <константное выражение>;
```

В дальнейшем в тексте программы вместо константного выражения достаточно указывать *имя*. Тогда при изменении программы достаточно исправить значение в одном месте при описании этой именованной константы, а компилятор внесет все исправления по всему тексту программы.
Пример.

```
const N = 10;
…
double a[N], b[2*N];
…
for (i = 0; i < N; i++)…
```

Здесь при описании размеров массивов и в операторе цикла используют одну и ту константу N. Если исправить строку описания на `const N = 20`, то компилятор автоматически исправит программу на обработку массивов в два раза больше.

По умолчанию именованные константы имеют тип целой константы `int`. Для констант других типов необходимо явно указывать их тип. Полный формат описания констант:

```
<тип> const <имя> = <константное выражение>;
```

Пример.

```
double const F = 10.234;
```

Помимо такой конструкции из С++ можно воспользоваться и классической конструкцией командой препроцессора `#define`:

```
#define <имя константы> <значение константы>;
```

Пример.

```
#define F 10.234;
```

Переопределение типа. В некоторых случаях удобно сложное описание типа, например структуры, обозначить коротким словом, а затем использовать это короткое обозначение в программе. Для этого можно использовать оператор задания типа `typedef`. Синтаксис команды следующий:

```
typedef <определение типа> <имя>;
```

В дальнейшем в тексте программы это имя можно использовать при описании типа.
Пример.

```
const N = 10;
typedef long double   real;
…
real a[N], b[2*N];
…
for (i = 0; i < N; i++)…
```

Здесь стандартный тип long double переобозначен идентификатором real, который потом и используется в программе. Здесь так же легко исправить тип описания нескольких переменных по всей программе, если изменить описание в операторе typedef.

Описание переменных в программе. Язык С (и С++) при написании программы допускает описание переменных по тексту программы, а не только в начале.
Пример.

```
for (int i = 0; i < 100; i++)
 {
  for (int j = 0; j < 200; j++)
    …
  …
 }
for (i = 0; i < 15; i++)
 {
  for (int j = 0; j < 25; j++)
    …
  …
 }
```

При этом необходимо помнить, что такое описание является локальным в том смысле, что оно известно только внутри блока, в котором описание было сделано. Поэтому в данном примере во второй группе вложенных циклов переменная i уже известна, а вот переменную j приходится описывать заново, так как действие описание прекратилось с закрытием блока.

Замечание. Необходимо быть осторожным и внимательным, так как блок присутствует независимо от того, есть скобки или нет. Так в приведенном примере если циклы по i содержат только цикл по j любой сложности, то операторные скобки могут быть опущены. Но блоки будут такие же, описание потребуется такое же, например:

```
for (int i = 0; i < 100; i++)
  for (int j = 0; j < 200; j++)
    …
for (i = 0; i < 15; i++)
  for (int j = 0; j < 25; j++)
    …
…
```

Комментарии. Для большей ясности и лучшего понимания текста программы она снабжается комментариями. Программы на языке С (и С++) могут содержать два вида комментариев. Для короткого комментария, поясняющего оператор можно использовать однострочный комментарий. Он начинается символом ‘//’ и действует до конца строки.
Пример.

```
double a, b[10], *c, *d[15];  // пример однострочного комментария
```

Для большого комментария можно использовать несколько однострочных комментариев или многострочный комментарий. Многострочный комментарий представляет собой текст, заключенный в парные скобки из двух символов ‘/*’ и ‘*/’.
Пример.

```
const N = 10;  /* это многострочный
комментарий, который заканчивается
закрывающей скобкой */
```

Структуры

Программа на языке С (и С++) при обработке сложных данных использует различные агрегированные типы данных, например массивы.

Для объединения разнородных данных базовых типов в единую переменную используют структуры. Фактически каждая структура это новый тип, который не описан в языке С (и С++).

Для описания новой структуры используют описание шаблона структуры. Описание шаблона выполняется по следующему правилу:

```
struct <имя шаблона> {
  тип1  имя1;
  тип2  имя2;
  …
  типN  имяN;
};
```

Имя шаблона является стандартным идентификатором и в дальнейшем может использоваться при описании переменных программы типа этой структуры. Переменные составляющие этот шаблон называются *полями*.

Пример.

```
struct man {
  int weight;
  int high, chest, waist, hip;
  double pay;
};

void main ()
{
  man ivan, peter;
  …
}
```

Здесь новый шаблон структуры `man`, а уже в программе используется этот шаблон для описания переменных типа структура `ivan, peter`. К полям переменной типа структуры обращение осуществляется указанием имени переменной и имени поля, отделенного точкой, например: `ivan.high = 170; peter.hip = 90;`.

Помимо переменных типа структуры в программе могут использоваться и указатели на тип структура. В этом случае поле от указателя отделяется парой ‘->’. Кроме этого можно выделять динамическую память под такую переменную или массив.

Пример.

```
void main ()
{
  man *gala, *people;
  …
  gala = new man;
  gala->weight = 50;
  people = new man [21];
  for (int i = 0; i < 21; i++)
    people [i]->pay = 120;
  …
  delete gala;
  delete people;
}
```

Как и для переменных других типов для структур можно описывать массивы. Полем структуры может быть переменная любого типа или указатель, массив или структура.

Одновременно при описании шаблона можно описывать и переменные. В этом случае описание выглядит как:

```
struct <имя шаблона> {
  тип1  имя1;
  тип2  имя2;
  …
  типN  имяN;
} <список имен переменных>;
```

Список имен представляет собой имена, разделенные запятой. Как и переменные других типов, так и переменные типа структура могут быть инициализированы при описании.

```
void main ()
```

```
{
  man funt = {65, 175, 90, 60, 90, 120};
  …
}
```

Помимо обычных структур есть два специализированных вида структур: объединение и битовая.

Подпрограммы

При разработке простых программ текст программы выступает как единое целое, оператор следует оператором, строка следует за строкой. Но при усложнении алгоритма решения его приходится разбивать на большие логические блоки. В частности, на блоки используемые часто, используемые повторно. В этих случаях прибегают к разбиению программы на подпрограммы.

В теории языков программирования различают два типа подпрограмм: *процедуры* и *функции*.

В отличие о других языков программирования, в языке С (и С++) используется только один тип подпрограмм — *функция*, то есть подпрограмма, которой через ее параметры передаются некоторые значения и которая возвращает результаты своей работы через значение функции (то есть, через имя функции) и специально выделенные для этой цели параметры.

Функция описывается *заголовком* и *телом*. *Заголовок* содержит:

- *имя функции*;
- *тип* возвращаемого результата;
- *типы* и *имена формальных параметров*, заключенные в круглые скобки и разделенные запятыми.

Телом функции является составной оператор, объединяющий *описание внутренних переменных* функции (или *локальные переменные*) и операторы, реализующие функцию:

```
<тип результата> <имя функции>(<список формальных параметров>)
{
<тело функции>
}
```

Пример.

```
double linefunc (double x, double a, double b)
{
  return (a*x + b);
}
```

В данном примере результат вычисления значения линейной функции возвращается через имя функции. То есть в программе такая функция может использоваться следующим образом:

```
double z = 15.2 + linefunc (2.3, 4.5, 6.7);
```

Другой пример описания и использования функции вычисления целочисленной степени целого числа.

Пример.

```
int pow (int x, int n)
{ /* возвращает степень n > 0 числа x */
 int i, p;
 p = 1;
 for (i = 1; i <= n; i++)
   p = p * x;
 return p;
}

main ()
{
 int i;
 for (i = 0; i < 10; i++)
   cout << i << ' ' << pow (2, i) << ' ' << pow (-3, i);
}
```

Фактические параметры, указываемые при вызове функции, могут задаваться константой, константным выражением, переменной или выражением.

Для обработки формальных параметров могут потребоваться дополнительные переменные, например, для организации цикла, подсчета промежуточного результата и т.п. Такие переменные называются *локальными переменными*, и описываются в теле функции по мере необходимости. В данном примере это переменная целого типа `i`.

Описание функции может следовать за функцией `main()` или находиться вообще в другом файле с исходным текстом или в библиотеке.

Для того чтобы компилятор мог выполнить *проверку* соответствия *типов* передаваемых аргументов *типам* формальных параметров в определении функции, до вызова функции нужно поместить *объявление* или *прототип* функции.

Правило. Прежде чем функция будет вызвана в программе, описание ее заголовка должно быть известно компилятору. То есть *описание* функции или ее *прототип* должны быть приведены выше или входить в состав *заголовочных файлов*, подключаемых к программе.

Прототип функции имеет такой же формат, что и определение функции с такой лишь разницей, что он не имеет тела функции, а заголовок функции кончается знаком “;”.

Определение. *Прототип* функции это *заголовок* функции, завершающийся символом точка с запятой ‘**;**’.

Прототип функции задает *имя* функции, *типы* и *число* формальных параметров, *тип* возвращаемого значения и *класс памяти*. Формальные параметры в прототипе могут иметь имена, но эти имена компилятор не использует. Поэтому в прототипе *имена* переменных *можно не писать*.

Пример.

```
double linefunc ();  // прототип функции без входных параметров
double linefunc (double, double, double);  // прототип функции с тремя
double linefunc (double x, double a, double b);  // входными параметрами
```

Здесь два последних прототипа совершенно эквивалентны.

Используемые в программе в операторе `#include` заголовочные файлы как раз и содержат описания стандартных функций, например, функции ввода–вывода или потокового ввода–вывода.

Компилятор использует *прототип* функции для сравнения *типов фактических параметров* при вызове функции с *типами формальных параметров*.

Примечание. Если *объявление* функции *не задано*, то по умолчанию, строится прототип на основании информации взятой из *первого вызова* функции. Однако такой прототип может некорректно представлять последующий вызов или определение функции. Поэтому *рекомендуется* ***всегда*** *задавать* прототипы функций или объявлять их до первого вызова.

Для реализации подпрограмм общего вида (процедур) в языке C (и C++) используются функции, *не возвращающие значение*. В поле определения типа результата таких функций необходимо писать ключевое слово `void`, указывающее, что функция не возвращает значение. А выход из такой подпрограммы осуществляется оператором `return` без параметров.

Предупреждение. Если функция *объявлена со спецификатором* типа возвращаемого значения, а фактически значение *не возвращается*, поведение вызывающей функции после возврата управления из такой функции, может быть *непредсказуемым*.

Список формальных параметров может быть пустым и описываться ключевым словом `void` (отсутствие параметров), а может заканчиваться и многоточием `(тип имя, ...)`. Это означает, что число аргументов функции переменно. Однако предполагается, что функция имеет, по крайней мере, столько аргументов, сколько формальных параметров задано перед последней запятой. Если передано больше значений, чем указано до последней запятой, то над такими аргументами не производится контроль типов и обработка их в подпрограмме выполняется специфичным образом.

Правило. При вызове функции необходимо соблюдать порядок и соответствие типов формальных параметров и фактических. Если тип формального параметра не указан, то по умолчанию — **int**.

При разработке программ функцию необходимо рассматривать как самостоятельный законченный фрагмент программы, предназначенный для решения некоторой небольшой подзадачи.

Передача параметров

Как правило, в языках программирования используют два механизма передачи параметров: *по значению* и *по ссылке*.

При передаче параметра *по значению* аргументом может быть произвольное *выражение*, значение которого передается в подпрограмму.

При передаче параметра *по ссылке* аргументом может быть только *переменная* (простая или структурированная). В этом случае в подпрограмму передается *не значение* переменной, а ее *адрес*, для того, чтобы по нему могло быть занесено новое значение передаваемой в качестве параметра переменной.

В языке С (и С++) все аргументы передаются *по значению*. Но синтаксис языка С (и С++) позволяет описать и использовать подпрограмму с параметрами, как бы передаваемых ‘по ссылке’.

Для организации передачи аргумента *по ссылке*, в заголовке подпрограммы такой формальный параметр описывается с использованием операции получения адреса объекта ‘&’. Тогда при вызове такой подпрограммы в подпрограмму передается не значение, а адрес требуемой переменной. В теле такой функции этому формальному параметру должно быть *присвоено значение* хотя бы один раз. Это значение и будет возвращено подпрограммой через этот формальный параметр.

Пример.

```
#include <iostream.h>

void subproc (double a, double b, double &c)
{
  c = a + b;  // вычисление выражения;
              // результат вернуть через формальный параметр 'c'
  return;
}

void main ()
{
  cout << "Start:"<< endl;
  double x = 5.0, y = 1.0, z;

  subproc (x, y, z);

  cout << "Исходные: " << x << ", " << y << endl;
  cout << "Результат = " << z << endl;
}  // end of program
```

Необходимо помнить, что при вызове функции в качестве фактического параметра, задаваемого на позиции формального параметра ‘по ссылке’, может использоваться *только переменная*. Ведь здесь компилятор должен подставить адрес, по которому можно укладывать значение; но ни константа, ни выражение принять значение на сохранение не может. В данном примере вызов функции можно выполнить так:

```
  subproc (5.0, 1.0, z);
```

Помимо операции получения адреса можно формальный параметр описывать и как указатель. При этом несколько изменяется работа с данным аргументом в самой подпрограмме: необходимо параметр описать как указатель и использовать, при доступе к нему, операции работы с указателями, что, естественно, немного усложняет текст.

Пример.

```
#include <iostream.h>
```

```
void subproc2 (double a, double b, double *c)
{
  // c = a + b;   неправильное использование параметра 'c'
  // b = 2 * c;  неправильное использование параметра 'c'

 *c = a + b;  // вычисление выражения;
               // результат вернуть через формальный параметр 'c'
 return;
}

void main ()
{
  cout << "Start:"<< endl;
  double x = 5.0, y = 1.0, z;

  subproc2 (x, y, &z);

  cout << "Исходные: " << x << ", " << y << endl;
  cout << "Результат = " << z << endl;
}  // end of program
```

Выполнение вызова функции осуществляется в следующем порядке:

1) вычисляются выражения входящие в список выражений фактических параметров; сравниваются типы результатов вычисленных выражений с типом соответствующим формальным параметрам, и если нет совпадения, то производится преобразование типов (проверяется столько аргументов, сколько задано).
2) заменяются формальные параметры на фактические;
3) передается управление на первый оператор тела процедуры;
4) производится вычисление и возврат по оператору `return`; если его нет, то вычисления производятся до последнего оператора и возвращаемое значение не определено.

Предупреждение. При написании тела функции желательно всегда указывать оператор `return`, а не надеяться на компилятор.

Функции обработки массивов

Функции могут обрабатывать не только простые переменные, но и массивы и структуры. В простейшем случае функция может разрабатываться для обработки массива (или двумерного) фиксированного размера.

Пример.

```
void print_m34 (int m[3][4])
{int i, j;
 for (i = 0; i < 3; i++)
  {
   for (j = 0; j < 4; j++)
     printf ("%d ", m [i][j]);
   printf ("\n");
  }
 return;
}
```

Такая функция будет выводить на экран только целочисленные матрицы размера 3×4.

При разработке функций естественно добиваться определенной универсальности. Поэтому функции обработки одномерных массивов, как правило, описывают на обработку массивов произвольной длины. Тогда в качестве дополнительного формального параметра необходимо указать целочисленную простую переменную, через которую будет задаваться размер этого массива (так как по-другому размер этого массива узнать нельзя).

Пример.

```
double summ (double m[], int n)
{double s;
 s = 0;
 for (int i = 0; i < n; i++)
  {s += m [i];
```

```
  }
 return s;
}
```

Заголовок этой функции можно описать и как:

```
double summ (double *m, int n)
```

Тело функции при этом будет точно таким же.

Такая функция будет суммировать массив любой длины, но только с элементами double. При этом в качестве фактического параметра можно указывать как имя массива так и адрес любого его элемента; длина может указываться как всего массива так и м*е*ньшая (но никак не б*о*льшая; иначе результат непредсказуем). Например, с помощью предыдущей функции можно найти сумму, как всего массива, так и его части.

Пример.

```
void main (void)
{double a [100];
 double s1, s2, s3, s4, s5;
  /* необходимо проинициализировать массив */
 s1 = summ (a, 100);  // сумма всего массива
 s2 = summ (&a[0], 100);  // тоже самое
 s3 = summ (&a[11], 33);  // сумма части массива
 s4 = summ (&a[0], 20) + summ (&a[20], 80); // сумма всего массива частями
 s5 = summ (a, 20) + summ (&a[20], 80); // тоже самое

 cout << s1 << " "
      << s2 << " "
      << s3 << " "
      << s4 << " "
      << s5 << endl;
}
```

Такая функция может обрабатывать, как и статически описанный массив, так и массив память которому выделялась динамически.

Функции обработки двумерных массивов

Так как в C (и C++) элементы матрицы располагаются по строкам, первая размерность не имеет отношения к задаче отыскания положения элементов. Так как строки матрицы размещаются в оперативной памяти последовательно друг за другом, то и обращаться к элементам матрицы можно как к элементам одномерного массива; надо лишь учитывать, что этот массив нарезан на порции размером, совпадающим с длинной строки матрицы. Поэтому при описании функции первый размер (количество строк) можно передавать через дополнительный параметр, чтобы знать, а сколько же там строк.

Пример.

```
void print_mN4 (int m[][4], int dim)
{int i, j;
 for (i = 0; i < dim; i++)
 {
  for (j = 0; j < 4; j++)
    printf ("%d ", m [i][j]);
  printf ("\n");
 }
 return;
}
```

Такая функция может корректно печатать любую матрицу, если длина ее строки равна 4.

Сложный случай возникает, когда требуется функция для обработки матрицы произвольного размера. Конечно, может показаться что достаточно передавать обе размерности. Но описать заголовок по аналогии с одномерным массивом не получается:

`void print_mNM (int m[][], int dim1, int dim2)` — запрещенная конструкция.

Простой выход или использовать матрицы, память которым выделяется динамически, или описывать функцию по обработке одномерного массива, но учитывающего, что он состоит из порций размером с длину строки.
Пример.

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

void print_mNM (int **m, int dim1, int dim2)
{int i, j;
 for (i = 0; i < dim1; i++)
  {
   for (j = 0; j < dim2; j++)
     printf ("%d ", m [i][j]);
   printf ("\n");
  }
 return;
}

const  nRow = 3, nCol = 2;

void main (void)
{clrscr ();

 int **a;
 a = new int* [nRow];

 for (int i = 0; i < nRow; i++)
  {a[i] = new int [nCol];
   for (int j = 0; j < nCol; j++)
     a[i][j] = выражение;
  }

 print_mNM (a, nRow, nCol);

 for (i = 0; i < nRow; i++)
   delete[] a[i];
 delete[] a;
}
```

Этот пример показывает, как можно обрабатывать динамически создаваемую матрицу, используя функцию.
Пример.

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

void print_mNM (int *m, int dim1, int dim2)
{int i, j;
 for (i = 0; i < dim1; i++)
  {
   for (j = 0; j < dim2; j++)
     printf ("%d ", m [i*dim2 + j]);
   printf ("\n");
  }
 return;
}

const  nRow = 3, nCol = 2;

void main (void)
{clrscr ();

 int a[3][2], *pa = &a[0][0];

 for (int i = 0; i < nRow; i++)
  {for (int j = 0; j < nCol; j++)
     a[i][j] = выражение;
  }

 print_mNM (pa, nRow, nCol);
```

```
}
```

Этот пример показывает, как можно обрабатывать статическую матрицу произвольного размера, используя функцию.

Эти функции рассчитаны на обработку динамически создаваемой матрицы и статически описанную матрицу. Но не наоборот: попытка переставить функции из этих примеров не даст результатов.

Эти функции могли бы не только брать значения из матрицы, но и изменять значения элементов матриц. То есть функции, которые формируют или изменяют элементы матрицы, могут иметь такое же описание матриц в заголовке.

Рекурсивные функции

Любая функция может вызывать другие функции. В том числе функция может вызывать сама себя. Такие функции называются *рекурсивными*. При этом необходимо соблюдать осторожность и помнить о следующих предостережениях:

- алгоритм функции должен быть таким, что цепочка вызовов функцией себя должна быть конечной; то есть при определенных условиях функция должна просто вернуть результат, а не вызывать себя снова;
- цепочка вызовов должна быть конечной и не очень глубокой; необходимо помнить, что при каждом новом вызове функции ее параметры и локальные переменные размещаются в стеке памяти задачи. Бесконечный вызов рекурсивной функции бессмыслен. Если рекурсивная функция может вызывать себя большое количество раз (имеет глубокую вложенность), то необходимо побеспокоится, чтобы компилятор построил программу с большим стеком памяти.

Классическим примером рекурсивной функции является функция вычисления факториала. Применение рекурсивной функции подсказывает одно из определений факториала: `n = (n-1)!×n`. Пример.

```
#include <iostream.h>
#include <conio.h>

int factor (int n)
{if (n == 0) return 1;  // факториал 0! = 1
 else return factor (n-1) * n;
}
void main (void)
{clrscr ();
 int m;
 cout << "Введите N: ";
 cin >> m;
 cout << "Факториал " << m << "! = " << factor (m);
}
```

Структура программы. Функция main ()

Программа может состоять из нескольких функций, которые могут размещаться в нескольких модулях (в нескольких файлах с исходным текстом) или библиотеках с объектным кодом. Поэтому помимо файлов с описанием функций проект должен содержать и прототипы функций размещенные, например, в заголовочных файлах. Головной модуль и другие исходные модули должны подключать заголовочные файлы через инструкцию препроцессора `#include`, в том числе и заголовочные файлы стандартных библиотек.

Среди всех функций составляющих проект программы должна быть функция с именем `main()`. Имя `main()` должно быть написано прописными (малыми) буквами. Как правило, в простых программах эта функция описывается без параметров и с типом результата `void`.

По сути, программа на языке C (и C++) состоит из нескольких функций, среди которых есть главная `main()`. В сложных проектах и функция `main()` может содержать параметры, через которые она получает значения флагов и параметров, с которыми была запущена программа в командной строке MS DOS.

Приложение 1

Задача № 1. Написать программу вычисления суммы элементов нескольких вещественных массивов с элементами типа double, используя подпрограмму.

Решение. Для правильного решения задачи необходимо ответить на несколько вопросов. Для удобства сведем эти вопросы в таблицу.

Таблица

Основные вопросы при решении задачи

№	Установка	Решение
1	Все массивы имеют элементы одного типа, задача единственная	Достаточно единственной подпрограммы при решении этой задачи
2	Требуется результат в виде единственного числа — суммы элементов	Результат можно вернуть через заголовок функции — число типа double
3	Для расчета суммы элементов массива достаточно иметь сам массив и знать количество элементов в нем	Достаточно только входных аргументов — ссылка на массив и количество элементов

В программе потребуется обрабатывать несколько массивов разной длины. Поэтому функцию желательно разработать не на конкретный массив, а попытаться написать некую универсальную для обработки произвольных массивов. Тип элементов массива произвольным быть не может, но можно обрабатывать массив произвольной длины, лишь бы эта длина была известна.

```
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>

double sumArr (double *arr, int num)
 {double sum;
  sum = 0;
  for (int i = 0; i < num; i++)
    sum += arr [i];  // накопление суммы элементов
  return sum;
 }
  /* Программа подсчета суммы элементов нескольких массивов
     и печати этих сумм на экран */

const Nf = 15;  // константа для указания размера массива f

void main ()
{     // описание и инициализация массивов
  double a [10] = {23.4, 17, 1.2, 3.141519265358, 18.01,
                   2.718281828, 9.8, 7.6, 5.40123};
  double b [5] = {6.3, 3.4, 7, 2.012345, 18.01};
  double c [2] = {50.05, 100.05};
  double d [200] = {50.2, 100.2, 150.2, 200.2, 250.2};  // остальные элементы
                                                        // инициализируются нулем
      // описание указателя на массив
  double *f;
      // описание переменных для хранения сумм
  double sA, sB, sC, sD, sF, sAB, sCD, sDF;  // переменные для хранения сумм
                                             // элементов массивов и
                                             // сумм от сумм элементов
  f = new double [Nf];  // выделение памяти под массив
  randomize ();  // инициализация датчика случайных чисел, чтобы
                 // при каждом новом выполнении задачи последовательность
                 // чисел генерировалась новая
  for (int i = 0; i < Nf; i++)
   {
```

```
    f [i] = random (20001);          // генерация чисел от 0 до 20000 (целых),
    f [i] = f [i] / 100.0 - 100.0;   // а затем пересчет их в диапазон -100 до 100
                                     // но уже вещественное и с дробной частью
   }
    // подсчет сумм, используя разработанную подпрограмму
  sA = sumArr (a, 10);
  sB = sumArr (b, 5);
  sC = sumArr (c, 2);
  sD = sumArr (d, 200);
  sF = sumArr (f, Nf);

  sAB = sumArr (a, 10) + sumArr (b, 5);
  sCD = sumArr (c, 2) + sumArr (d, 200);
  sDF = sumArr (d, 200) + sumArr (f, Nf);

  clrscr ();
  cout << "Start:"<< endl;

  cout << " Сумма элементов массива a = " << sA << endl;
  cout << " Сумма элементов массива b = " << sB << endl;
  cout << " Сумма элементов массива c = " << sC << endl;
  cout << " Сумма элементов массива d = " << sD << endl;
  cout << " Сумма элементов массива f = " << sF << endl;
  cout << " Сумма элементов массива a и b = " << sAB << endl;
  cout << " Сумма элементов массива c и d = " << sCD << endl;
  cout << " Сумма элементов массива d и f = " << sDF << endl;

  cout << " Сумма всех элементов массивов a, b, c, d и f = "
       << sumArr (a, 10) + sumArr (b, 5) + sumArr (c, 2) + sumArr (d, 200) +  +
sumArr (f, Nf);

  delete[] f;  // главный оператор программы
}  // end of program
```

Для примера в данной задаче приведена обработка массивов различной длины. Кроме статических массивов в программе описывается указатель и динамически выделяется память под массив. Любой из этих массивов может быть обработан написанной подпрограммой.