Глава 1

Работа с массивами

Очень часто в процессе программирования возникают ситуации, когда необходимо накапливать и обрабатывать большие объемы информации и использование стандартных типов данных нерационально. Для таких целей используются составные типы данных, т. е., типы, состоящие из стандартных типов данных.

В данной главе будет рассмотрен один из простейших составных типов — массив. Массив — это именованная структура, которая содержит элементы одного типа. Также будет рассмотрен еще один составной тип данных — указатели, которые хранят адрес переменной определенного типа.

1.1. Указатели

До этого в программе всегда объявлялись простые переменные. Оператор объявления определяет тип и символическое имя переменной, а также требует от программы выделить память под эту переменную и определяет скрытым образом ее местоположение. Для того, чтобы определить адрес простой переменной необходимо применить операцию *взятия адреса*: `&`⟨*имя переменной*⟩.

Следующая программа демонстрирует операцию взятия адреса для переменной типа `int`:

Листинг 1.1. Операция взятия адреса переменной

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5   int a = 5;
6   cout << "value=" << a << " address=" << &a << endl;
7   return 0;
8 }
```

Результат работы программы:

```
value=5 address=0026FE40
```

Результат работы программы — это адрес ячейки памяти, в которой расположен первый байт, занимаемый этой переменной. Обычно операция взятия адреса дает результат в шестнадцатиричной нотации.

Таким образом, значение переменной воспринимается как именованная величина, а ее адрес — как производная. Однако, во многих языках программирования существуют специальные типы переменных, которые в качестве именованной величины хранят не значение переменной, а ее адрес. Такие переменные называются *указателями*. Для получения значения, находящегося в ячейке памяти, адрес которой хранит указатель, используется операция *разыменования*: `*`⟨*имя переменной*⟩. Следующая программа демонстрирует операцию разыменования для указателя `p`:

Листинг 1.2. Пример операции разыменования типов

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5   int a = 5;
6   int *p = &a;
7   cout << "address: " << p << " value: " << *p <<endl;
8   return 0;
9 }
```

Результат работы программы:

```
address=002EF8D4 value=5
```

Как видно из листинга 1.2 объявление указателя имеет следующий вид:

⟨*тип*⟩ `*`⟨*имя переменной*⟩

Например, запись `int *p` означает, что значение переменной `*p` имеет тип `int`, а `p` — это указатель на тип `int`.

Переменная-указатель никогда не бывает просто указателем, она всегда указывает на какой-нибудь тип. Указатели могут указывать на различные типы данных, имеющие разный размер, но сами указатели в памяти обычно занимают 2 или 4 байта.

Необходимо помнить, что при объявлении указателей автоматически выделяется память только для записи адреса, но выделения памяти под хранение значения, расположенного по этому адресу, не происходит.

Если указатель не инициализировать, то в поле для записи адреса будет записано случайное шестнадцатиричное число, которое может означать любую ячейку памяти, в том числе, занятую системными данными. Попытка изменить данные по этому адресу может привести к непредсказуемым последствиям.

Способов инициализации несколько:

- Как показано в листинге 1.2, сначала инициализируется переменная определенного типа, затем объявляется указатель на адрес этой переменной:

```
int a = 5; //переменная типа int

int *p = &a //указатель на адрес переменной типа int
```

- Указателю присваивается адрес в явном виде, но необходимо точно знать, что нужная ячейка памяти свободна:

```
int *p;

p = (int *)0x002EF8D4; //указан адрес ячейки памяти
```

- Используется нулевой указатель (null-указатель). Стандарт языка C++ гарантирует, что нулевой указатель никогда не указывает на корректные данные, поэтому он часто используется в качестве признака окончания каких-либо действий:

```
int *p = NULL; //null-указатель или

int *p = 0;
```

- Используется операция выделения памяти new. С помощью операции выделения памяти, память выделяется не на стадии компиляции, а на стадии выполнения программы, что позволяет экономить память при условии, если размер и количество данных заранее неизвестны. Общий вид:

$$\langle тип \rangle * \langle имя\ указателя \rangle = \texttt{new}\ \langle тип \rangle$$

Поскольку память выделяется программистом на стадии выполнения программы, то и освобождается память тоже программистом с помощью операции delete .

Например, объявляется указатель на тип int и выделяется память под переменную типа int, выполняются какие-либо действия и освобождается память:

Листинг 1.3.

```
int main(){
  ...
  int *p = new int;
  //операторы
  delete p;
  ...
}
```

1.2. Одномерные массивы

Массив — это структура данных, которая содержит множество значений, относящихся к одному и тому же типу. Тип может быть любым, как базовым, так и составным. Каждое значение сохраняется в отдельном элементе массива, и компьютер сохраняет все элементы в памяти последовательно, друг за другом.

Существует два способа представления массивов: *статические*, когда память под элементы массива выделяется на стадии компиляции и необходимо заранее знать размер массива, и *динамические*, когда память выделяется программистом с помощью команды `new` на стадии выполнения программы.

Рассмотрим сначала статические массивы.

1.2.1. Статические одномерные массивы

Статический одномерный массив — это последовательность элементов одного типа, размер которого должен быть известен заранее. Каждое значение хранится в отдельном элементе, и в памяти компьютера элементы массива располагаются последовательно, один за другим.

Для создания массива должно быть известно три составляющих: тип элементов, имя массива и количество элементов в массиве.

Формат объявления массива:

⟨*тип*⟩ ⟨*имя массива*⟩ `[`⟨*размерность массива*⟩`]`.

В отличии от остальных случаев, в этом представлении квадратные скобки являются обязательными, а не обозначает необязательные элементы.

Например, `int a[5]` — массив целых чисел, состоящих из пяти элементов. К каждому элементу массива можно обратиться с помощью индекса (порядковый номер элемента), т. е., первый элемент массива — `a[0]`, второй — `a[1]`, последний — `a[4]`.

Нумерация элементов массива начинается с НУЛЯ. Следовательно, последний элемент массива, состоящего из `n` элементов, имеет индекс `a[n-1]`. Язык C++ не проверяет правильность вводимого индекса элемента массива, поэтому возможны ошибки при попытке работы с несуществующим элементом, например, вызывается элемент `a[5]` массива, который содержит всего 5 элементов (от `[0]` до `a[4]`). Такая проверка остается за программистом. Об этом необходимо помнить при работе с массивами.

С массивами можно работать только ПОЭЛЕМЕНТНО. Их нельзя присвоить друг другу, с массивами нельзя выполнять никаких операций, можно работать только с элементами массивов.

Возможно несколько вариантов объявления и инициализации массива:

1. С помощью одновременно объявления и инициализации массива:

   ```
   int a[5] = {1, 3, 8, 9, 7};
   ```

   Массив содержит 5 элементов:

   $\{1,\ 3,\ 8,\ 9,\ 7\}$.

2. Можно инициализировать только часть элементов, тогда остальным элементам будет присвоено значение 0:

   ```
   int a[5] = {1, 2, 3};
   ```

   Массив содержит 5 элементов:

   $\{1,\ 2,\ 3,\ 0,\ 0\}$.

   Тогда можно легко обнулить элементы массива:

   ```
   int a[5] = {0};
   ```

   Первому элементу будет присвоено значение 0, следуя операции присваивания, остальным — по умолчанию.

3. Если оставить квадратные скобки пустыми и инициализировать массив, то компилятор подсчитает количество элементов самостоятельно:

```
int a[] = {1, 2, 3, 5};
```

Результатом будет массив, состоящий из четырех элементов. Не стоит использовать этот способ слишком часто, так как компилятор может определить размер массива непредсказуемо.

4. Заполнить массив поэлементно, используя оператор цикла и вводя элементы самостоятельно:

```
for (int i = 0; i < n; i++){
    cout << "a["<< i << "]=";
    cin >> a[i];
}
```

5. Заполнить массив поэлементно, используя оператор псевдослучайных чисел:

Листинг 1.4. Использование генератора псевдослучайных чисел

```
1  #include<iostream>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <time.h>
5  using namespace std;
6
7  int main(){
8    int a[10], n;
9    cout << "n="; cin >> n;                  // число эпементов массива
10   srand ((unsigned int)time(NULL));    // начальная точка генерации
11   for (int i = 0; i < n; i++){
12     a[i] = rand() % 15;                  //псевдослучайное число
13     cout <<"a[" << i << "]=" << a[i] << endl;
14   }
15   return 0;
16 }
```

Результат работы программы:

```
n = 8
a[0] = 3
a[1] = 6
a[2] = 9
a[3] = 11
```

```
a[4] = 6
a[5] = 12
a[6] = 9
a[7] = 8
```

Функция `rand()` определяет псевдослучайное число из диапазона `[0, MAX_RAND]` (`MAX_RAND=32767`). Элементом массива в листинге 1.4 является остаток от деления соответствующего числа на 15. Для вызова функции `rand()` необходима стартовая точка. Она генерируется с помощью функции `srand()`. Если не использовать эту функцию, то при каждом следующем запуске программы будет генерироваться та же самая последовательность чисел, что и при первом запуске. Функция `time(NULL)` определяет количество секунд, прошедших с 01.01.1970 до текущего времени процессора. Естественно, что это время при новом запуске будет другим, следовательно, будет сгенерирована другая последовательность.

1.2.2. Динамические одномерные массивы

При работе со статическими массивами размер массива должен быть определен на стадии компиляции, т. е., если для решения какой-то задачи заранее определили, что массив состоит из 200 элементов, а реально используется только 10, следовательно, будет выделена излишняя память, и наоборот, если изначально определили, что массив состоит из 10 элементов, а используется 200, то произойдет ошибка выполнения. Поэтому проще работать с динамическими массивами, память под которые выделяется на этапе выполнения программы.

Создать динамический массив достаточно просто с помощью операции `new`. Для этого необходимо создать указатель определенного типа, и, используя `new`, указать тип элементов и количество таких элементов в квадратных скобках:

```
int *mas = new int [10];
```

В данном примере указатель `mas` возвращает адрес нулевого элемента массива, а всего память выделяется под 10 элементов типа `int`.

Если массив был создан с помощью операции `new`, соответственно необходимо очистить память с помощью операции `delete`:

```
delete [] mas;
```

Квадратные скобки перед указателем означают, что необходимо очистить всю память, выделенную под массив, а не только удалить нулевой элемент.

Общая форма выделения и присваивания памяти для динамического массива выглядит следующим образом:

⟨*тип*⟩ *⟨*имя указателя*⟩ = new ⟨*тип*⟩ [⟨*число элементов*⟩].

Вызов операции new выделяет достаточно большой блок памяти, необходимый для того, чтобы в нем уместилось `число элементов` элементов типа `тип` и устанавливает в `имя указателя` указатель на нулевой элемент массива.

Как обращаться к элементам динамического массива? Элементы массива в памяти компьютера расположены подряд, поэтому возможны два варианта: через *арифметику указателей* и через *индексирование массива*.

Рассмотрим арифметику указателей. Указатель определяет адрес нулевого элемента массива, предположим типа int. Если сдвинуть указатель на 4 байта, то он будет указывать уже на первый элемент массива. Арифметика указателей в данном случае заключается в том, что увеличение указателя на единицу, означает сдвиг на столько байт, сколько занимает элемент соответствующего типа (см. рисунок 1.1).

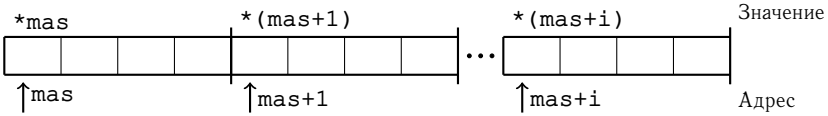

Рис. 1.1. Схематическое представление одномерного массива в памяти компьютера

Пример 1.1. Дан динамический массив типа int, состоящий из пяти элементов:

Листинг 1.5. Работа с динамическим массивом

```
1 #include<iostream>
2 using namespace std;
3
4 int main (){
5   int n = 5, i = 1;
6   int *mas = new int [n];
7   int *p;
8   p = (mas + n);                    // указатель на адрес памяти,
```

```
9   cout << "adress p=" << p << endl;       // следующий за
10  while (mas != p){                       // последним элементом
11    *mas = i;                            // значение элемента массива
12    cout << "mas[" << i-1 << "]=" << *mas << endl;
13    cout<< "adress " << mas <<endl;
14    mas++;                               // указатель на следующий элемент
15    i++;                                 // увеличиваем значение i
16  }
17  mas = mas - n;                          //возврат указателя на 0-ой элемент
18  delete [] mas;
19  return 0;
20 }
```

Результат работы программы:

```
adress p = 00141DA4
mas[0] = 1
adress p = 00141D90
mas[1] = 2
adress p = 00141D94
mas[2] = 3
adress p = 00141D98
mas[3] = 4
adress p = 00141D9C
mas[4] = 5
```

В строке 6 листинга 1.5 объявлен массив типа `int`, состоящий из пяти элементов. Указатель `p`, объявленный в строке 7 и равный `mas + n` (строка 8), определяет адрес памяти, следующий за последним элементом массива. До тех пор, пока `mas` не равен `p`, выполняется цикл (строки 10 — 16):

1. Элементу, на который указывает `mas`, присваивается значение `i` (строка 11).

2. Выводится значение элемента и его адрес на экран (строки 12–13).

3. Увеличивается указатель на единицу, т. е., адрес увеличивается на 4 байта (строка 14).

4. Увеличивается значение `i` на единицу (строка 15).

После окончания цикла указатель `mas` возвращается на нулевой элемент, т. е., адрес уменьшается на `n * sizeof(int)` байт (строка 17), и очищается память, выделенная под массив (строка 18).

Как можно видеть в результате, элементы массива действительно располагаются последовательно, каждый элемент занимает 4 байта. □

Таким образом, к `i`-тому элементу массива можно обратиться следующим образом: `*(mas+i);`

Скобки обязательны, поскольку приоритет операции разыменования (*) выше, чем операции сложения. Поэтому запись `*mas+i` означает, что к значению текущего элемента `mas` добавляется значение `i`, а запись `*(mas+i)` означает, что сначала указатель сдвигается на `i * sizeof(`⟨*mun*⟩`)` байт, а потом используется значение элемента, на который указывает указатель после сдвига.

Рассмотрим теперь работу с массивами через индексацию массивов. В данном случае, работа с динамическим массивом ничем не отличается от работы со статическим, только имя указателя используется как имя массива:

Листинг 1.6. Индексация одномерного динамического массива

```
1  #include<iostream>
2  using namespace std;
3
4  int main (){
5    int n = 5, i = 1;
6    int *mas = new int [n];
7    for (int i = 0; i < n; i++) {
8      mas[i] = i + 1;          //заполняем i элемент
9      cout << "mas[" << i << "]=" << mas[i] << endl;
10     cout<< "adress " << &(mas[i]) <<endl;
11   }
12   delete [] mas;
13   return 0;
14 }
```

В строке 9 листинга 1.6 на экран выводится значение `i`-ого элемента массива, в строке 10 — выводится адрес элемента с помощью операции взятия адреса `(&)`.

Как можно легко проверить, результат работы программы листинга 1.6 будет таким же, как и в случае листинга 1.5, за исключением адресов памяти (при повтор-

ном запуске может быть выделен другой участок памяти).

Таким образом, массивы и указатели взаимозаменяемы. На самом деле, в случае работы с массивами с помощью индексов, при запуске программы сначала происходит неявный переход к указателям, а потом доступ к элементам массива происходит с использованием библиотеки указателей.

То есть, работа с массивами проще для понимания и чтения кода, работа с указателями быстрее по времени исполнения.

1.3. Двумерные массивы

Рассмотрим теперь работу с двумерными массивами. Это массив, который удобно представлять в виде таблицы, состоящей из строк и столбцов. Тогда элемент массива с индексами `i` и `j` находится на пересечении `i`-ой строки и `j`-ого столбца массива. Двумерные массива также бывают статические и динамические.

1.3.1. Статические двумерные массивы

Описание статического двумерного массива в общем случае имеет вид:

⟨*тип*⟩ ⟨*имя массива*⟩`[`⟨*число строк*⟩`][`⟨*число столбцов*⟩`]`.

Например, запись вида `int mas[10][10];` означает, что создан массив, состоящий из 10 строк и 10 столбцов, элементы которого целые числа.

Работа с массивами происходит поэлементно. Так, для того, чтобы создать массив, заполнить его и вывести на экран, необходимо использовать вложенные циклы:

Листинг 1.7. Пример работы с двумерным статическим массивом

```
#include<iostream>
using namespace std;

int main(){
  int mas[100][100];     //массив из 100 строк и 100 столбцов
  int n;
  cout << "n="; cin >> n; //ввод размерности массива
  /*----------------ввод массива-------------------*/
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++){
```

```
11        cout << "mas[" << i << "][" << j << "]=";
12        cin >> mas[i][j];
13      }
14    /*---------------вывод массива-------------------*/
15    cout << "Array:\n";
16    for (int i = 0; i < n; i++, cout << endl)
17      for (int j = 0; j < n; j++)
18        cout << mas[i][j] << "  ";
19    return 0;
20  }
```

Результат работы программы:

```
n = 3
mas[0][0]=1
mas[0][1]=2
mas[0][2]=3
mas[1][0]=4
mas[1][1]=5
mas[1][2]=6
mas[2][0]=7
mas[2][1]=8
mas[2][2]=9
matrix:
1 2 3
4 5 6
7 8 9
```

Ввод вручную элементов двумерного массива занимает достаточно времени и этим способом лучше пользоваться только в случае, когда необходимо ввести массив специфическим образом, например, все строки состоят из одинаковых чисел и т. д. Лучше пользоваться генератором псевдослучайных чисел, по аналогии с одномерным массивом (см. раздел 1.2.1):

Листинг 1.8. Ввод двумерного массива с помощью генератора псевдослучайных чисел

```
1 #include<iostream>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
using namespace std;

int main (){
  int a[10][10],n;
  cout << "n="; cin >> n;                  // число эпементов массива
  srand ((unsigned)time(NULL));           // начальная точка генерации
  for (int i = 0;  i < n; i++, cout << endl)
    for (int j = 0; j < n; j++){
      a[i][j] = rand() % 15;            //псевдослучайное число
      cout << a[i][j] << " ";
    }
  return 0;
}
```

Результат работы программы:

```
n = 5
0 12 14 7 12
14 1 7 0 6
1 0 1 2 8
5 1 11 9 7
3 4 13 12 12
```

Как видно из листингов 1.7–1.8 для объявления двумерного статического массива необходимо сразу определить количество строк и столбцов массива. Следовательно, память выделяется под массив существенно большего размера, чем реально необходимо.

На самом деле двумерный статический массив в памяти представляется как массив одномерный: сначала записываются элементы нулевой строки, потом первой и т. д. Получается, что память расходуется нерационально, поэтому лучше использовать динамические двумерные массивы, память под который выделяется на этапе работы программы самим программистом.

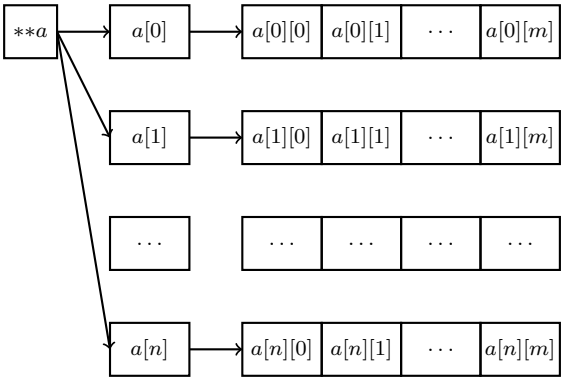

Рис. 1.2. Представление двумерного динамического массива в памяти компьютера

1.3.2. Двумерные динамические массивы

Динамический двумерный массив расположен в памяти более рационально. Как видно на рисунке 1.2, двумерный динамический массив можно трактовать как набор отдельных одномерных динамических массивов, которые могут быть расположены в любом свободном месте памяти, необязательно подряд. При таком подходе каждая строка двумерного массива может содержать разное количество элементов, что абсолютно неприемлемо для статического массива.

Другими словами, динамический двумерный массив — это указатель на массив указателей, каждый из которых в свою очередь указывает на одномерный динамический массив.

Работать с двумерными динамическими массивами также можно через арифметику указателей и через индексирование. Арифметика указателей для двумерного массива достаточно сложена, поэтому в будущем будем пользоваться индексами. Арифметику указателей можно рассмотреть самостоятельно.

Объявление двумерного динамического массива имеет свои особенности. Так как речь идет об указателях, то сначала с помощью оператора `new` выделяется память под массив указателей:

⟨*тип*⟩ `**`⟨*имя массива*⟩ `=` `new` ⟨*тип*⟩ `*[`⟨*число строк*⟩`].`

Затем для каждого элемента одномерного массива указателей выделяется память под соответствующий массив:

```
for(i = 0; i < ⟨число строк⟩; i++)
    ⟨имя массива⟩[i] = new ⟨тип⟩ [⟨число столбцов⟩].
```

И только после этого можно заполнять массив. Основное преимущество использования двумерного динамического массива — это возможность задавать размерность массива в ходе выполнения программы, а не определять ее заранее.

Ниже приведен пример объявления и инициализации двумерного динамического массива:

Листинг 1.9. Пример работы с двумерным динамическим массивом

```
#include<iostream>
#include<time.h>
#include<stdlib.h>
using namespace std;

int main(){
  int n, m;
  cout << "n = "; cin >> n;       //размерность массива
  cout << "m = "; cin >> m;
  int **a = new int *[n];         //выделение памяти под массив указателей
  for (int i = 0; i < n; i++)
    a[i] = new int [m];          //выделение памяти для каждого указателя
  srand((unsigned)time(NULL));
  for (int i = 0; i < n; i++, cout << endl)
    for (int j = 0; j < m; j++){
      a[i][j] = rand() % 15;      //заполнение элемента массива
      cout << a[i][j] << " ";     //вывод на экран
    }
   return 0;
}
```

Если запустить листинг 1.9, то можно увидеть, что результат работы программы будет похожим на результат работы со статическим двумерным массивом (листинг 1.8).

Примером работы с динамическим двумерным массивом, каждая строка которого содержит различное количество элементов, может служить треугольник Паскаля.

1.4. Функции и массивы

1.4.1. Одномерные массивы

Все рассмотренные выше примеры функций использовали только значения базовых типов данных. Однако функции могут служить инструментами и для обработки более сложных типов, таких как массивы и структуры.

Допустим у нас есть некий талантливый программист, работающий сразу в четырех организациях. У него есть данные по каждой организации, сколько он там в какой месяц заработал. Теперь он хочет узнать суммарную, максимальную и минимальную сумму, полученную им в каждом месяце. В листинге 1.10 приведен соответствующий код.

Листинг 1.10.

```
#include <iostream>
using namespace std;

int main(){
  double work1[12], work2[12], work3[12], work4[12], max[12], min[12], sum[12];
  cout << "Введите данные по первой работе\n";
  for (int i = 0; i < 12; i++){
    cout << i + 1 << "-й месяц - ";
    cin>>work1[i];
  }
  cout << "Введите данные по второй работе\n";
  for (int i = 0; i < 12; i++){
    cout << i + 1 << "-й месяц - ";
    cin >> work2[i];
  }
  cout << "Введите данные по третьей работе\n";
  for (int i = 0; i < 12; i++){
    cout << i + 1 << "-й месяц - ";
    cin >> work3[i];
  }
  cout << "Введите данные по четвертой работе\n";
  for (int i = 0; i < 12; i++){
    cout << i + 1 << "-й месяц - ";
    cin >> work4[i];
```

```
25  }
26  for (int i = 0; i < 12; i++){
27    max[i] = work1[i] > work2[i] ? work1[i] : work2[i];
28    if (max[i] < work3[i]) max[i] = work3[i];
29    if (max[i] < work4[i]) max[i] = work4[i];
30    min[i] = work1[i] < work2[i] ? work1[i] : work2[i];
31    if (min[i] > work3[i]) min[i] = work3[i];
32    if (min[i] > work4[i]) min[i] = work4[i];
33    sum[i] = work1[i] + work2[i] + work3[i] + work4[i];
34  }
35  cout << "Суммарная зарплата\n";
36  cout << "_____\n";
37  cout << "|  1|  2|  3|  4|  5|  6|  7|  8|  9| 10| 11| 12|\n";
38  cout << "_____\n";
39  cout << '|';
40  for (int i = 0; i < 12; i++){
41    cout.width(5);
42    cout << sum[i] << '|';
43  }
44  cout << endl;
45  cout << "_____\n";
46  cout << "максимальная зарплата\n";
47  cout << "_____\n";
48  cout << "|  1|  2|  3|  4|  5|  6|  7|  8|  9| 10| 11| 12|\n";
49  cout << "_____\n";
50  cout << '|';
51  for (int i = 0; i < 12; i++){
52    cout.width(5);
53    cout << max[i] << '|';
54  }
55  cout << endl;
56  cout << "_____\n";
57  cout << "минимальная зарплата\n";
58  cout << "_____\n";
59  cout << "|  1|  2|  3|  4|  5|  6|  7|  8|  9| 10| 11| 12|\n";
60  cout << "_____\n";
61  cout << '|';
62  for (int i = 0; i < 12; i++){
63    cout.width(5);
```

```
64      cout << min[i] << '|';\
65    }
66    cout << endl;
67    cout << "_____\n";
68    return 0;
69  }
```

В результате работы для некоторого набора входных данных программа, например, выведет:

```
Суммарная зарплата
```

	1		2		3		4		5		6		7		8		9		10		11		12
	3		4		5		6		7		8		9		10		11		12		13		14

```
максимальная зарплата
```

	1		2		3		4		5		6		7		8		9		10		11		12
	1		1		2		3		4		5		6		7		8		9		10		11

```
минимальная зарплата
```

	1		2		3		4		5		6		7		8		9		10		11		12
	0		1		1		1		1		1		1		1		1		1		1		1

Очевидно, что использование функций для ввода и вывода элементов массива сильно уменьшило бы размеры программ и облегчило работу программиста.

Итак, требуется создать функцию, в которую можно было бы передать массив в качестве параметра. Возьмем следующее объявление функции:

`void input_arr (char* title, int arr[ ], int n).`

Выглядит вполне прилично. Квадратные скобки указывают на то, что `arr` — массив, а тот факт, что они пусты, говорит о том, что эту функцию можно применять с массивами любого размера. Но бывает, что некоторые вещи не являются тем, чем они кажутся: `arr` — на самом деле не массив, а указатель! Однако этот факт никак не повлияет на код программы, его можно писать так, как если бы `arr` все-таки

был массивом. Во-первых, убедимся, что такой подход работает, а потом разберемся, почему он работает.

В листинге 1.11 приведен код решаемой задачи, но с использованием функций ввода и вывода. Запустив его легко убедиться, что при том же наборе входных данных результат будет идентичный уже приведенному. Только код став в полтора раза меньше.

Листинг 1.11.

```
#include <iostream>
using namespace std;

void printline (int n){
  for (int i = 0; i < n; i++)
    cout << "_______";
  cout << '_' << endl;
 }

void input_arr (char* title, int arr[ ], int n){
  cout << title << endl;
  for (int i = 0; i < n; i++){
    cout << i << "-й месяц - ";
    cin >> arr[i];
  }
}

void output_arr (char* title, int arr[ ], int n){
  cout << title << endl;
   printline (n);
  cout << '|';
  for (int i = 0; i < n; i++){
    cout.width(5);
    cout << i + 1 << '|';
  }
  cout << endl;
   printline (n);
  cout << '|';
  for (int i = 0; i < n; i++){
    cout.width(5);
```

```
31      cout << arr[i] << '|';
32    }
33    cout << endl;
34     printline (n);
35  }
36
37  int main(){
38    int work1[12], work2[12], work3[12], work4[12], max[12], min[12], sum[12];
39    input_arr("Введите данные по первой работе", work1, 12);
40    input_arr("Введите данные по второй работе", work2, 12);
41    input_arr("Введите данные по третьей работе", work3, 12);
42    input_arr("Введите данные по четвертой работе", work4, 12);
43    for (int i = 0; i < 12; i++){
44      max[i] = work1[i] > work2[i] ? work1[i] : work2[i];
45      if (max[i] < work3[i]) max[i] = work3[i];
46      if (max[i] < work4[i]) max[i] = work4[i];
47      min[i] = work1[i] < work2[i] ? work1[i] : work2[i];
48      if (min[i] > work3[i]) min[i] = work3[i];
49      if (min[i] > work4[i]) min[i] = work4[i];
50      sum[i] = work1[i] + work2[i] + work3[i] + work4[i];
51    }
52    output_arr("Суммарная зарплата", sum, 12);
53    output_arr("Максимальная зарплата", sum, 12);
54    output_arr("Минимальная зарплата", sum, 12);
55    return 0;
56  }
```

Заметим, что C++, в большинстве контекстов трактует имя массива как указатель, т.е. имя массива `arr == &arr[0]` рассматривается как адрес его первого элемента.

В листинге 1.11 присутствует следующий вызов функции:

```
input_arr ("Введите данные по первой работе", work1, 12);
```

Здесь `work1` — имя массива, поэтому, согласно правилам C++, `work1` представляет собой адрес первого элемента этого массива. То есть функции передается адрес. Поскольку массив имеет тип элементов `int`, `work1` должно иметь тип указателя на `int`, или `int *`. Это предполагает, что корректный заголовок функции должен быть таким:

```
void input_arr (char *title, int *arr, int n).
```

Здесь `int *arr` заменяет `int arr[ ]`. На самом деле оба варианта заголовка корректны, потому что в C++ нотации `int *arr` и `int arr[ ]` имеют идентичный смысл, когда применяются в заголовке или прототипе функции (и только в этом случае).

Теперь понять, как работает эта программа не представляет труда для тех, кто внимательно ознакомился с разделом «массивы».

Рассмотрим, что следует из листинга 1.11. Вызов функции

```
output_arr("Суммарная зарплата", sum, 12);
```

передает адрес первого элемента массива `sum` и количество его элементов в функцию `output_arr`. Функция `output_arr` присваивает адрес `sum` переменной-указателю `arr` , а значение 12 — переменной `n` типа `int`. Это значит, что в листинге 1.11 на самом деле в функцию не передается содержимое массива. Вместо этого программа сообщает функции, где находится массив (то есть сообщает его адрес), каков тип его элементов (тип массива) и сколько в нем содержится элементов (переменная `n`). Вооруженная этой информацией, функция затем использует исходный массив. Если передается обычная переменная, то функция работает с ее копией. Но если передается массив, то функция работает с его **оригиналом**.

Такой подход к обработке массива экономит время и память, необходимые для копирования всего массива. Накладные расходы, связанные с использованием таких копий, могли быть весьма ощутимыми при работе с большими массивами. С копиями программам понадобилось бы не только больше компьютерной памяти, но и больше времени, чтобы копировать крупные блоки данных. Кроме того появляется возможность использовать некоторые нестандартные обращения. Допустим массив `sum` заполнен так, что каждый элемент равен порядковому номеру месяца, т. е. `sum [0] = 1` и т. д. Тогда можно вывести не весь массив, а только его часть. Например обращение `output_arr("", sum, 5);` выведет на экран следующую таблицу:

```
_____________________________
|    1|    2|    3|    4|    5|
_____________________________
|    1|    2|    3|    4|    5|
_____________________________
```

А если использовать следующее обращение `output_arr("", sum + 4, 5) ;` то результат будет выглядеть так:

```
|    1|    2|    3|    4|    5|
|    5|    6|    7|    8|    9|
```

Это произойдет потому, что `sum+4` в качестве указателя начала массива в функцию передаст указатель на его 5-й элемент.

Работа с двумерными массивами мало отличается от работы с одномерными массивами. Ведь и в этом случае имя массива трактуется как его адрес, поэтому соответствующий формальный параметр является указателем — так же , как и в случае одномерного массива. Только двумерный массив — это как-бы массив массивов, и поэтому при обращении к его конкретному элементу придется встретиться с двумя указателями. Рассмотрим это на простом примере.

Следующие два обращения к элементу массива `ar[5][4]` идентичны:

```
ar[r][c] == *( *( ar + r)+ c)
```

Чтобы понять это, нужно разобрать выражение по частям, начиная изнутри:

```
ar //указатель на первую строку - массив из 4 int
ar + r //указатель на строку r (также массив из 4 int)
* (ar + r)//сама строка r (сам массив из 4 int,
          //т.~е. указатель на первое число в ней - ar[r])
* (ar + r)+ c //указатель на элемент int под номером c
          //в строке r, т.~е. ar[r] + c
* ( *(ar + r)+ c //значение int под номером c
          //в строке r, т.~е. ar[r][c]
```

Таким образом следующие описания функций будут идентичны:

```
void input_array (int arr[][], int col, int row);
void input_array (int **arr, int col, int row);
void input_array (int ( *arr)[], int col, int row);
```

Для работы с двумерным динамическим массивом для облегчения чтения кода необходимы как минимум две функции: объявление и инициализация массива и вывод массива на экран.

Функция объявления и инициализации массива представлена в листинге 1.12:

Листинг 1.12. Функция ввода двумерного массива с клавиатуры

```
int **create (int n, int m){ //возвращает указатель на первый элемент
  int **a = new int *[n];
  /*******выделение памяти***********/
  for (int i = 0; i < n; i++)
    a[i] = new int [m];
  /****заполнение массива с экрана***/
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++){
      cout << "a[" << i << "][" << j << "]=";
      cin >> a[i][j];
    }
  return a;
}
```

Функция вывода массива на экран представлена в листинге 1.13:

Листинг 1.13. Функция вывода массива на экран

```
void print (int **a, int n, int m){ //функция выводит массив на экран
  for (int i = 0; i < n; i++, cout << endl)
    for (int j = 0; j < m; j++)
      cout << a[i][j] << " ";
}
```