ГЛАВА 10

Массивы и указатели

В этой главе:

- Ключевое слово: `static`
- Операции: `& *` (унарные)
- Создание и инициализация массивов
- Указатели (на основе сведений, которые вам уже известны) и какое отношение они имеют к массивам
- Написание функций, обрабатывающих массивы
- Двумерные массивы

Люди обращаются за помощью к компьютеру при решении таких задач, как подсчет ежемесячных расходов, расчет ежедневного количества осадков, ежеквартальных продаж, еженедельного прироста веса и так далее. Предприятия обращаются к помощи компьютера при составлении платежных ведомостей, при учете материально-производственных запасов и при расчетах с заказчиками. Будучи программистом, вы неизбежно имеете дело с большими объемами соответствующих данных. Довольно часто массивы предлагают наилучшие способы манипулирования такими данными — удобные и эффективные. В главе 6 было введено понятие массива, а в этой главе мы изучим массивы более подробно. В частности, мы рассмотрим функции, выполняющие обработку массивов. Такие функции предоставляют вам возможность распространить преимущества модульного программирования на массивы. Изучая материал этой главы, вы сами можете убедиться в том, насколько тесно связаны между собой массивы и указатели.

Массивы

Вспомните, что *массив* образует некоторая последовательность элементов одного и того же типа данных. Вы используете *объявления* с целью уведомить компилятор о том, что вам необходим массив. *Объявление массива* сообщает компилятору, сколько элементов содержит массив и какой тип имеют его элементы. Располагая такого рода информацией, компилятор может правильно создать массив. Элементы массива могут иметь те же типы, что и обыкновенные переменные. Рассмотрим следующий пример объявления массива:

```
/* несколько объявлений массивов */
int main(void)
{
   float candy[365];      /* массив из 365 значений типа float */
   char code[12];         /* массив из 12 значений типа char */
   int states[50];        /* массив из 50 значений типа int */
   ...
}
```

Квадратные скобки (`[]`) свидетельствуют о том, что `candy` и другие такие же структуры данных являются массивами, а число, заключенное в квадратные скобки, задает количество элементов в массиве.

Чтобы получит доступ к элементам массива, вы должны указать отдельный элемент, используя для этой цели его номер, который также называется *индексом*. Нумерация элементов массива начинается с 0. Следовательно, `candy[0]` — это первый элемент массива `candy`, а `candy[364]` — 365-й, он же последний, элемент массива.

Все это нам уже известно; продолжим далее изучение массивов.

Инициализация

Массивы часто используются для хранения данных, необходимых программе. Например, каждый элемент 12-элементный массива может содержать количество дней в соответствующем месяце. В случаях, подобных данному, удобно выполнить инициализацию массива в начале программы. Посмотрим, как это делается. Вы уже знаете, как выполняется инициализация однозначных переменных (иногда они называются *скалярными*) в объявлениях с выражениями наподобие

```
int fix = 1;
float flax = PI * 2;
```

в которых, как можно надеяться, константа была определена ранее. Язык C расширяет инициализацию на массивы с помощью новых синтаксических средств, как показано ниже:

```
int main(void)
{
   int powers[8] = {1,2,4,6,8,16,32,64};   /* только в стандарте ANSI */
   ...
}
```

Нетрудно видеть, что вы инициализируете массив с применением списка элементов, отделенных друг от друга запятыми, заключенного в квадратные скобки. Между запятыми и значениями при желании можно вставлять пробелы. Первому элементу (`powers[0]`) присваивается значение `1` и так далее. (Если ваш компилятор отказывается выполнять такую форму инициализации, рассматривая ее как синтаксическую ошибку, значит, компилятор был разработан до появления стандарта ANSI. Проблема решается путем помещения перед объявлением массива ключевого слово `static`. В главе 12 значение этого ключевого слова обсуждается более подробно.) В листинге 10.1 показана короткая программа, которая выводит на печать количество дней каждого месяца.

Листинг 10.1. Программа `day_mon1.c`

```
/* day_mon1.c -- выводит на печать количество дней каждого месяца */
#include <stdio.h>
#define MONTHS 12
int main(void)
{
   int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   int index;
   for (index = 0; index < MONTHS; index++)
      printf("Месяц %d имеет %2d дней (день).\n", index + 1,
             days[index]);
   return 0;
}
```

Выходные данные имеют следующий вид:

```
Месяц 1 имеет 31 дней (день).
Месяц 2 имеет 28 дней (день).
Месяц 3 имеет 31 дней (день).
Месяц 4 имеет 30 дней (день).
Месяц 5 имеет 31 дней (день).
Месяц 6 имеет 30 дней (день).
Месяц 7 имеет 31 дней (день).
Месяц 8 имеет 31 дней (день).
Месяц 9 имеет 30 дней (день).
Месяц 10 имеет 31 дней (день).
Месяц 11 имеет 30 дней (день).
Месяц 12 имеет 31 дней (день).
```

Не ахти какая программа, однако, она ошибается только один раз в четыре года. Программа инициализирует массив `days[]` с использованием списка значений, отделенных друг от друга запятыми, заключенного в квадратные скобки.

Обратите внимание на то, что в этом примере присутствует символическая константа `MONTHS` для представления размера массива. Это распространенная и рекомендованная практика. Например, если вдруг мир перейдет на 13-месячный календарь, достаточно будет внести соответствующее изменение в оператор `#define`, и не нужно будет выискивать в программе все места, в которых используется размер массива.

Использование констант в массивах

Иногда вам приходится использовать массив, предназначенный только для чтения. То есть программа извлекает значения из массива, но не предпринимает попыток записывать новые значения в этот массив. В таких случаях вы можете и должны использовать ключевое слово `const` во время объявления и инициализации массива. С учетом сказанного выше, в программе, представленной в листинге 10.1, лучше воспользоваться следующей конструкцией:

```
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Это позволяет программе рассматривать каждый элемент массива как константу. Как и в случае обычных переменных, вы должны использовать объявление для инициализации данные типа `const`, поскольку если они объявлены как `const`, вы не можете впоследствии присваивать им новые значения. Теперь, когда мы знаем об этом, мы можем использовать константы в последующих примерах.

Что произойдет, если вы не сможете инициализировать массив? Об этом мы узнаем после исследования программы из листинга 10.2.

Листинг 10.2. Программа `no_data.c`

```
/* no_data.c -- неинициализированный массив */
#include <stdio.h>
#define SIZE 4

int main(void)
{
   int no_data[SIZE];      /* неинициализированный массив */
   int i;

   printf("%2s%14s\n",
          "i", "no_data[i]");
   for (i = 0; i < SIZE; i++)
      printf("%2d%14d\n", i, no_data[i]);

   return 0;
}
```

Вот один из примеров выходных данных этой программы (результаты могут меняться):

```
i   no_data[i]
0           16
1      4204937
2      4219854
3   2147348480
```

Элементы массива мало чем отличаются от обычных переменных, и если вы не инициализируете их, они могут принимать произвольные значения. Компилятор использует те значения, которые уже были записаны в соответствующих ячейках памяти, вот почему ваши результаты могут отличаться от представленных выше.

Учет различных классов памяти

Массивы, как и все другие переменные, можно создавать с использованием *классов памяти*. Эта тема рассматривается в главе 12, однако, сейчас вам достаточно знать, что в текущей главе описаны массивы, которые относятся к автоматическому классу памяти. Это означает, что они объявлены внутри функции без употребления ключевого слова `static`. Все переменные и массивы, использовавшиеся до сих пор в этой книге, относятся к автоматическому классу. Мы упомянули здесь классы памяти по той простой причине, что время от времени различные классы памяти проявляют различные свойства, так что вы не должны автоматически переносить все сказанное в этой главе на другие классы памяти. В частности, переменные и массивы некоторых не рассмотренных здесь классов памяти, не будучи инициализированными, получают значение 0.

Количество элементов в списке должно соответствовать размеру массива. Но что будет, если вы ошибетесь при подсчете? Попробуем еще раз выполнить последний пример, как показано в листинге 10.3, включив список, который в два раза короче объявленного размера массива.

Листинг 10.3. Программа `somedata.c`

```
/* some_data.c -- частично инициализированный массив */
#include <stdio.h>
#define SIZE 4

int main(void)
{
   int some_data[SIZE] = {1492, 1066};
   int i;

   printf("%2s%14s\n",
          "i", "some_data[i]");
   for (i = 0; i < SIZE; i++)
      printf("%2d%14d\n", i, some_data[i]);

   return 0;
}
```

На этот раз выходные данные приобретают следующий вид:

```
i   some_data[i]
0           1492
1           1066
2              0
3              0
```

Нетрудно убедиться в том, что у компилятора проблем не было. Как только значения в списке закончатся, остальные элементы он инициализирует нулями. Иначе говоря, если вы вообще не выполняете инициализацию массива, его элементы, подобно обычным переменным, получают произвольные значения мусора в памяти, в то же время, если вы выполняете частичную инициализацию массива, оставшимся неинициализированным элементам присваиваются нулевые значения.

Но компиляторы не простят вам ошибку, если вы укажете список, в котором значений больше, чем размер объявленного массива. Такая чрезмерная щедрость рассматривается как ошибка. Тем не менее, нет необходимости выставлять себя мишенью для насмешек со стороны собственного компилятора. Вместо этого вы можете предоставить компилятору возможность привести в соответствие размер массива со списком, исключив размер из квадратных скобок (листинг 10.4).

Листинг 10.4. Программа `day_mon2.c`

```
/* day_mon2.c -- компилятор сам подсчитывает количество элементов */
#include <stdio.h>

int main(void)
{
   const int days[] = {31,28,31,30,31,30,31,31,30,31};
   int index;

   for (index = 0; index < sizeof days / sizeof days[0]; index++)
      printf("Месяц %2d имеет %d дней (день).\n", index +1,
             days[index]);

   return 0;
}
```

В листинге 10.4 необходимо отметить два момента:

- Когда вы освобождаете квадратные скобки с целью инициализации массива, компилятор проводит подсчет элементов списка и устанавливает размер массива равным этому числу.
- Обратите внимание на то, как был скорректирован управляющий оператор цикла `for`. Не имея уверенности в том, что нам удастся выполнить этот подсчет правильно (что вполне объяснимо), мы предоставляем право компьютеру самостоятельно определить размер массива. Операция `sizeof` возвращает размер в байтах объекта, или *типа*, следующего за ним. Таким образом, `sizeof days` — это размер в байтах всего массива, а `sizeof days[0]` — размер в байтах одного элемента этого массива. Выполнив деление размера всего массива на размер одного элемента, мы узнаем, сколько элементов содержится в массиве.

Ниже показан результат выполнения этой программы:

```
Месяц 1 имеет 31 дней (день).
Месяц 2 имеет 28 дней (день).
Месяц 3 имеет 31 дней (день).
Месяц 4 имеет 30 дней (день).
Месяц 5 имеет 31 дней (день).
Месяц 6 имеет 30 дней (день).
Месяц 7 имеет 31 дней (день).
Месяц 8 имеет 31 дней (день).
Месяц 9 имеет 30 дней (день).
Месяц 10 имеет 31 дней (день).
```

Вот как! Вывелось лишь 10 значений, а использованный нами метод, позволяющий программе самостоятельно определить размер массива, не предоставил возможности распечатать массив до конца. Это подчеркивает потенциальный недостаток автоматизированного подсчета: ошибочное количество элементов может оказаться необнаруженным.

Существует еще один более короткий метод инициализации массивов. Однако, поскольку его применение возможно только в отношении символьных строк, мы отложим его рассмотрение до следующей главы.

Выделенные инициализаторы (стандарт C99)

Стандарт C99 добавляет в язык C новую возможность — *выделенные инициализаторы*. Это свойство позволяет выбирать, какие элементы должны быть инициализированы. Предположим, например, что вы хотите инициализировать только последний элемент массива. Полагаясь только на традиционные синтаксические средства инициализации языка C, вы должны также инициализировать и все элементы, предшествующие последнему:

```
int arr[6] = {0,0,0,0,0,212};         // традиционный синтаксис
```

В условиях действия стандарта C99 вы можете использовать индекс в квадратных скобках в списке инициализации при описании некоторого конкретного элемента:

```
int arr[6] = {[5] = 212}; // инициализировать элемент arr[5] значением 212
```

Как и при обычной инициализации, после того, как вы выполните инициализацию, по меньшей мере, одного элемента, инициализированные элементы получают значение 0. В листинге 10.5 представлен более сложный пример.

Листинг 10.5. Программа `designate.c`

```
// designate.c -- использование выделенных инициализаторов
#include <stdio.h>
#define MONTHS 12

int main(void)
{
   int days[MONTHS] = {31,28, [4] = 31,30,31, [1] = 29};
   int i;

   for (i = 0; i < MONTHS; i++)
      printf("%2d %d\n", i + 1, days[i]);

   return 0;
}
```

Ниже приведены выходные данные для случая, когда компилятор поддерживает это свойство стандарта C99:

```
1   31
2   29
3    0
4    0
5   31
6   30
7   31
8    0
9    0
10   0
11   0
12   0
```

Эти выходные данные отражают два важных свойства выделенных инициализаторов. Во-первых, если за выделенным инициализатором следует код с дальнейшими значениями, как, например, в последовательности `[4] = 31,30,31`, при этом приведенные значения используются для инициализации последующих элементов. То есть, после инициализации `days[4]` значением `31` этот код инициализирует элементы `days[5]` и `days[6]`, соответственно, значениями `30` и `31`. Во-вторых, если этот код инициализирует конкретный элемент некоторым значением более одного раза, в действие вступает последняя инициализация. Например, в листинге 10.5, начало списка инициализации инициализирует элемент `days[1]` значением `28`, но это значение будет позже перекрыто выделенным инициализатором `[1] = 29`.

Присваивание значений массивам

После того, как массив будет объявлен, вы можете *присвоить* значения элементам массива, используя для этого *индекс* элемента массива. Например, следующий фрагмент программного кода присваивает (или назначает) массиву четные числовые значения:

```
/* присваивание значений массиву */
#include <stdio.h>
#define SIZE 50
int main(void)
{
   int counter, evens[SIZE];
   for (counter = 0; counter < SIZE; counter++)
      evens[counter] = 2 * counter;
   ...
}
```

Обратите внимание, что в этом коде используется цикл для поэлементного присваивания значений. Язык C не позволяет присваивать один массив в качестве значения другого как элемента данных. Нельзя также использовать форму списка, заключенного в фигурные скобки, нигде, кроме как в операторе инициализации.

В показанном ниже фрагменте программного кода демонстрируются некоторые формы недопустимых методов присваивания значений.

```
/* Недопустимые методы присваивания значений элементам массива */
#define SIZE 5
int main(void)
{
   int oxen[SIZE] = {5,3,2,8};     /* здесь все в порядке */
   int yaks[SIZE];
   yaks = oxen;                    /* недопустимый оператор */
   yaks[SIZE] = oxen[SIZE];        /* неправильно */
   yaks[SIZE] = {5,3,2,8};         /* этот метод не работает */
```

Границы массива

Вы должны убедиться в том, что используемые индексы элементов массива не выходят за допустимые пределы, другими словами, вы должны убедиться в том, что они имеют значения, разрешенные для данного массива. Например, предположим, что вы сделали следующее объявления:

```
int doofi[20];
```

В этом случае вы должны следить за тем, чтобы программа использовала индексы в диапазоне от 0 до 19, поскольку компилятор не станет делать эту проверку за вас.

Рассмотрим программу в листинге 10.6. Она создает массив из четырех элементов, а затем безответственно использует значения индексов в диапазоне от –1 до 6.

Листинг 10.6. Программа `bounds.c`

```
// bounds.c -- выход за границы массива
#include <stdio.h>
#define SIZE 4
int main(void)
{
   int value1 = 44;
   int arr[SIZE];
   int value2 = 88;
   int i;
```

```
    printf("value1 = %d, value2 = %d\n", value1, value2);
    for (i = -1; i <= SIZE; i++)
        arr[i] = 2 * i + 1;
    for (i = -1; i < 7; i++)
       printf("%2d %d\n", i , arr[i]);
    printf("value1 = %d, value2 = %d\n", value1, value2);
    return 0;
}
```

Компилятор не выполняет проверку правильности использования индексов в массиве. В стандартном языке C результат неправильного использования индекса не определен. Это означает, что когда вы инициируете программу, может показаться, что она работает правильно, вести себя странным образом или аварийно завершиться. Ниже показан пример выходных данных этой программы, откомпилированной с помощью Digital Mars 8.4:

```
value1 = 44, value2 = 88
-1 -1
0  1
1  3
2  5
3  7
4  9
5  5
6  1245120
value1 = -1, value2 = 9
```

Обратите внимание, что компилятор сохранил значение `value2` непосредственно после массива, а значение `value1` — непосредственно перед ним. (Другие компиляторы могут хранить данные в памяти в другом порядке.) В данном случае `arr[-1]` соответствует той же ячейке памяти, что и `value1`, а `arr[4]` — той же ячейке памяти, что и `value2`. В силу этого обстоятельства, использование индексов элементов массива, выходящих за допустимые пределы, приводит к тому, что программа меняет значения других переменных. Другой компилятор может выдать другие результаты, и одним из них может быть аварийное завершение программы.

Вы, должно быть, хотели бы знать, почему язык C допускает такие вещи. Все это является следствием философии языка C, которая предоставляет программистам большую свободу в принятии решений. Отказ от проверки выхода за допустимые пределы ускоряет выполнение программы на C. Компилятор не всегда способен отлавливать все индексные ошибки, поскольку значение индекса может оставаться неопределенным до тех пор, пока не начнется выполнение результирующей программы. В силу этого обстоятельства, чтобы обеспечить безопасность, компилятор должен добавить в программу дополнительный код для проверки каждого индекса во время выполнения, но от этого замедлится само выполнение программы. По этой причине C оставляет за программистом задачу правильного кодирования, и вознаграждает его за это увеличением быстродействия программы. Разумеется, не все программисты заслуживают такого доверия, вот тогда-то и начинаются настоящие проблемы.

Следует всегда помнить одну простую истину — нумерация элементов массива начинается с 0. Нужно также выработать в себе привычку использовать символические константы в объявлениях массивов и других местах, где используется размер массива:

```
#define SIZE 4
int main(void)
{
   int arr[SIZE];
   for (i = 0; i < SIZE; i++)
   ....
```

Это позволит обрести уверенность в том, что вы последовательно используете один и тот же размер массива в программе.

Указание размера массива

До сих пор при объявлении массивов использовались целочисленные константы:

```
#define SIZE 4
int main(void)
{
   int arr[SIZE];         // символическая целочисленная константа
   double lots[144];      // литеральная целочисленная константа
   ...
```

Что еще разрешено программисту? До появления стандарта C99 на этот вопрос можно было ответить, что при объявлении массива вы можете воспользоваться *константным целочисленным выражением*, заключив его в квадратные скобки. Константное выражение — это выражение, сформированное из целочисленных констант. В этом смысле выражение `sizeof` рассматривается как целочисленная константа, в то же время (в отличие от `case` в языке C++) значением типа `const` не является. Кроме того, значение такого выражения должно быть больше 0:

```
int n = 5;
int m = 8;
float a1[5];                // да
float a2[5*2 + 1];          // да
float a3[sizeof(int) + 1];  // да
float a4[-4];               // нет, размер должен быть > 0
float a5[0];                // нет, размер должен быть > 0
float a6[2.5];              // нет, размер должен быть целым числом
float a7[(int)2.5];         // да, преобразование типа из float int constant
float a8[n];                // не разрешалось до появления стандарта C99
float a9[m];                // не разрешалось до появления стандарта C99
```

Как показывают комментарии, компиляторы языка C, действующие в соответствие с требованиями стандарта C90, не разрешают двух последних объявлений. В то же время стандарт C99 их допускает, благодаря чему создается новый тип массивов, получивший название *массива переменной длины*.

Стандарт C99 вводит в употребление массивы переменной длины главным образом для того, чтобы увеличить возможности языка C в плане применения численных методов. Например, массивы переменной длины значительно облегчают преобразование существующих библиотек программ на языке FORTRAN, выполняющих цифро-

вые расчеты, в программы на языке C. На массивы переменной длины накладываются определенные ограничения, например, вы не можете выполнять инициализацию массива переменной длины в его объявлении. Мы еще вернемся к массивам переменной длины в этой главе несколько позже, после того, как изучим ограничения, которым в языке C подвергается классический массив.

Многомерные массивы

Мисс Темпест Клауд (Tempest Cloud — буквально, "грозовая туча"), специалист-метеоролог, исключительно серьезно относящаяся к своим профессиональным обязанностям, намеревается провести анализ данные о ежемесячных осадках за последние пять лет. Одно из ее первых решений касается выбора подходящей формы представления данных. Один из возможных вариантов состоит в использовании 60 переменных, по одной для каждого элемента данных. (Мы уже обсуждали этот вариант, нам он и сейчас кажется таким же бессмысленным, каким он казался и раньше.) Использование массива, содержащего 60 элементов, намного лучше, в то же время гораздо удобнее держать отдельно все данные, относящиеся к тому или иному конкретному году. Можно использовать пять массивов по 12 элементов, но такая попытка имеет свои недостатки и перерастает в трудно решаемую проблему, если мисс Темпест Клауд решит изучить данные о ежемесячных осадках за период длиной в 50 лет вместо пяти лет. Для этого ей потребуется более подходящая форма представления данных.

Более рациональный подход предусматривает использование массива массивов. Главный массив должен содержать пять элементов, по одному на каждый год. Каждый из этих элементов, в свою очередь, представляет собой 12-элементный массив, по одному элементу на каждый месяц. Такой массив объявляется следующим образом:

```
float rain[5][12];   // массив, состоящий из 5 массивов, каждый из
                     // которых состоит из 12 переменных типа float
```

Один из подходов к состоит в том, что сначала рассматривается внутренняя часть объявления (выделенная полужирным):

```
float rain[5][12];   // rain - это массив из 5 пока еще не известных объектов
```

Это говорит о том, что `rain` является массивом, состоящим из пяти элементов. Но что представляет собой каждый из этих элементов? Теперь рассмотрим другую часть этого объявления (выделенную полужирным):

```
float rain[5][12];   // массив из 12 значений типа float
```

Это говорит о том, что каждый элемент массива имеет тип `float[12]`, то есть каждый из этих пяти элементов массива `rain` сам по себе является массивом из 12 значений типа `float`.

В соответствие с этой логикой элемент `rain[0]`, будучи первым элементом массива `rain`, представляет собой массив из 12 значений типа `float`. Это справедливо и в отношении элементов `rain[1]`, `rain[2]` и так далее. Если `rain[0]` есть массив, его первым элементом будет `rain[0][0]`, вторым элементом — `rain[0][1]` и так далее. Короче говоря, `rain` — это пятиэлементный массив 12-элементных массивов значений типа `float`, `rain[0]` — массив из 12 элементов типа `float`, а `rain[0][0]` — значение типа `float`. Для получения доступа, скажем, к значению, находящемуся в строке 2 и столб-

це 3, используется конструкция `rain[2][3]`. (Напоминаем, что отсчет элементов массива начинается с 0, так что строка с номером 2 будет третьей.)

Вы можете рассматривать этот массив `rain` как двумерный массив, состоящий из пяти строк, при этом в каждой строке имеется 12 столбцов, как показано на рис. 10.1. Изменяя второй индекс, вы продвигаетесь по строке, месяц за месяцем. Изменяя первый индекс, вы перемещаетесь вертикально вдоль столбца, год за годом.

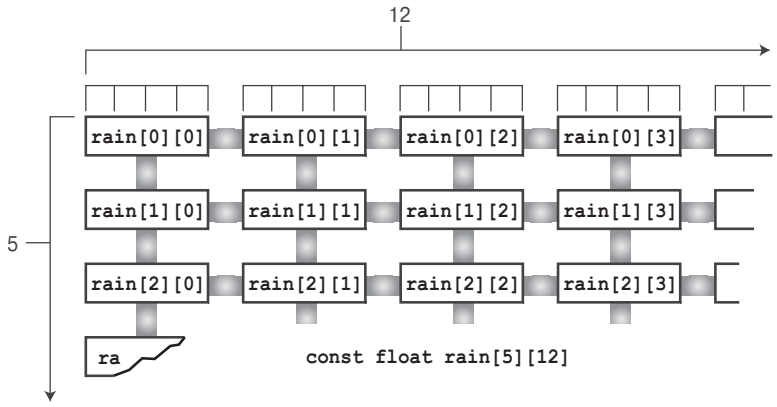

Рис. 10.1. *Двумерный массив*

Двумерное представление — это всего лишь удобный способ просмотра массива с двумя индексами. В памяти компьютера такой массив хранится последовательно, начиная с первого 12-элементного массива, за которым следует второй 12-элементный массив, и так далее.

Воспользуемся таким двумерным массивом в программе обработки погодных данных. Цель этой программы состоит в подсчете осадков за год, в вычислении средних ежегодных осадков и средних ежемесячных осадков. Чтобы вычислить общие осадки за год, нужно просуммировать все данные конкретной строки. Чтобы вычислить осадки за конкретный месяц, потребуется сложить все значения в заданном столбце. Двумерный массив упрощает визуализацию и выполнение этих действий. В листинге 10.7 показана соответствующая программа.

Листинг 10.7. Программа `rain.c`

```
/* rain.c -- вычисляет итоговые данные по годам, ежегодные средние значения
            и ежемесячные средние значения осадков за период в несколько лет*/
#include <stdio.h>
#define MONTHS 12  // количество месяцев в году
#define YEARS 5    // количество лет, в течение которых проводились наблюдения
int main(void)
{
   // инициализация массива данными об осадках за период с 2000 по 2004
   const float rain[YEARS][MONTHS] =
   {
      {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
      {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
      {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
```

```
      {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
      {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
   };
   int year, month;
   float subtot, total;

   printf(" ГОД КОЛИЧЕСТВО ОСАДКОВ (в дюймах)\n");
   for (year = 0, total = 0; year < YEARS; year++)
   {      // для каждого года суммарное количество осадков за каждый месяц
      for (month = 0, subtot = 0; month < MONTHS; month++)
         subtot += rain[year][month];
      printf("%5d %15.1f\n", 2000 + year, subtot);
      total += subtot;           // общая сумма за все годы
   }
   printf("\nСреднегодовое количество осадков составляет %.1f дюймов.\n\n",
           total/YEARS);
   printf("СРЕДНЕМЕСЯЧНОЕ КОЛИЧЕСТВО ОСАДКОВ:\n\n");
   printf(" Янв  Фев  Мар  Апр  Май  Июн  Июл  Авг  Сен  Окт");
   printf(" Ноя  Дек\n");
   for (month = 0; month < MONTHS; month++)
   {      // суммарные осадки по каждому месяцу на протяжении всего периода
      for (year = 0, subtot =0; year < YEARS; year++)
         subtot += rain[year][month];
      printf("%4.1f ", subtot/YEARS);
   }
   printf("\n");

   return 0;
}
```

Ниже показаны выходные данные этой программы:

```
ГОД КОЛИЧЕСТВО ОСАДКОВ (в дюймах)
2000            32.4
2001            37.9
2002            49.8
2003            44.0
2004            32.9
Среднегодовое количество осадков составляет 39.4 дюймов.

MONTHLY AVERAGES:

Янв  Фев  Мар  Апр  Май  Июн  Июл  Авг  Сен  Окт  Ноя  Дек
7.3  7.3  4.9  3.0  2.3  0.6  1.2  0.3  0.5  1.7  3.6  6.7
```

Во время изучения этой программы обратите особое внимание на инициализации и на схему вычислений. При этом инициализация является достаточно сложной процедурой, поэтому сначала рассмотрим более простую часть (вычисления).

Чтобы вычислить итоговую сумму за год, значение `year` остается неизменным, в то время как значение `month` пробегает весь диапазон значений. Это внутренний цикл `for` первой части программы. Затем этот процесс выполняется для следующего значения переменной `year`. Это внешний цикл первой части программы. Структура вложенного цикла вполне естественна при работе с двумерными циклами. Один цикл выполняют обработку первого индекса, а второй цикл обрабатывает второй индекс.

Вторая часть программы имеет аналогичную структуру, но на этот раз значение year меняется во внутреннем цикле, а значение month — во внешнем. Вспомните, что каждый раз, когда внешний цикл выполняет одну итерацию, внутренний цикл пробегает весь диапазон значений. По этой причине, при такой организации цикл пробегает все года, прежде чем изменится месяц. Сначала вы получаете среднегодовое значение осадков для первого месяца, затем для второго и так далее.

Инициализация двумерного массива

В основу инициализации двумерного массива положена методика инициализации одномерного массива. Во-первых, вспомните, что инициализация одномерного массива выполняется следующим образом:

```
sometype ar1[5] = {val1, val2, val3, val4, val5};
```

В рассматриваемом случае значения val1, val2 и так далее соответствуют некоторому типу sometype. Например, если бы sometype был типом int, значение val1 могло бы быть 7, или если бы sometype был бы типом double, значение val1 могло бы быть 11.34. Однако rain — это массив, состоящий из пяти элементов, причем этими элементами являются элементы в виде массивов, каждый из которых содержит 12 значений типа float. Следовательно, что касается массива rain, то в качестве val1 должно быть значение, пригодное для инициализации одномерного массива значений типа float, например, следующий массив:

```
{4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6}
```

То есть, если sometype — массив, состоящий из 12 значений типа double, то значение val1 представляет собой список 12 значений типа double. Следовательно, для инициализации двумерного массива, такого как rain, нам нужен список из пяти таких объектов, отделенных друг от друга запятыми:

```
const float rain[YEARS][MONTHS] =
{
   {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
   {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
   {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
   {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
   {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
};
```

Такая инициализация требует задания пяти списков числовых значений, заключенных в фигурные скобки. Данные, содержащиеся в первой паре фигурных скобок, присваиваются первой строке массива, данные, содержащиеся во второй паре фигурных скобок, — второй строке массива и так далее. Рассмотренные выше правила, касающиеся несоответствия размеров данных и массивов, применимы к каждой строке. Иначе говоря, если внутренняя пара фигурных скобок содержит 10 чисел, только 10 элементов получат соответствующие значения. В этом случае два последних элемента в этой строке инициализируются нулями. Если в списке слишком много чисел, это означает ошибку, и эти числа не будут распределены в следующей строке.

Можно опустить внутренние фигурные скобки и оставить две внешних скобки. Если в этом случае в скобках будет представлено корректное количество элементов, результат будет таким же. Однако, если элементов меньше, чем необходимо, массив заполняется последовательно, строка за строкой, пока не закончатся данные.

После этого оставшиеся элементы инициализируются нулями. На рис. 10.2 показаны оба способа инициализации массива.

Поскольку массив `rain` содержит данные, которые не могут быть изменены, программа использует модификатор `const` в объявлении массивов.

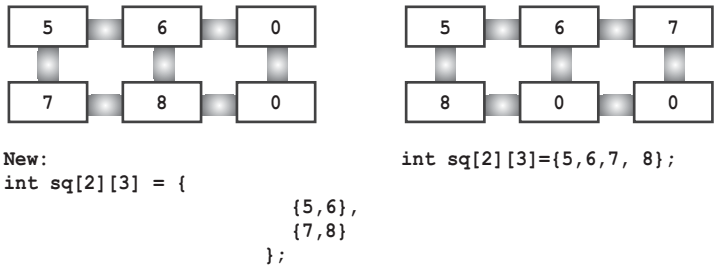

Рис. 10.2. *Два метода инициализации массива*

Массивы с размерностями больше двух

Все, что было сказано о двумерных массивах, можно распространить на трехмерные массивы и на массивы больших размерностей. Вы можете объявить трехмерный массив следующим образом:

```
int box[10][20][30];
```

Вы можете рассматривать одномерный массив как строку данных, двумерный массив — как таблицу данных, а трехмерный массив — как набор таблиц данных. Например, вы можете рассматривать многомерную таблицу как совокупность 10 двумерных массивов (размером 20×30 каждый), помещенных друг поверх друга.

Другим способом восприятия `box` является массив массивов, состоящих из массивов. Иначе говоря, это массив, элементы которого представляют собой 20-элементный массив. Каждый элемент этого 20-элементного массива представляет собой 30-элементный массив. Либо вы просто можете рассматривать массивы в терминах количества необходимых индексов.

Как правило, для обработки трехмерных массивов используется три вложенных цикла, для обработки четырехмерных массивов — циклы с глубиной вложения, равной четырем, и так далее. В наших примерах в основном мы будем пользоваться двумерными массивами.

Указатели и массивы

Указатели, как следует из главы 9, представляют собой символический способ использования адресов. Поскольку аппаратные инструкции вычислительных машин в значительной степени зависят от адресов, указатели позволяют формулировать свои намерения достаточно близко к тому, как машина выражает свои задачи. Это соответствие повышает эффективность программ, использующих указатели. В частности, указатели позволяют эффективно работать с массивами. В самом деле, как вы увидите далее, система обозначения массивов просто представляет собой особый вид использования указателей.

Примером такого особого использования может служить тот факт, что имя массива является также адресом первого элемента массива. Другими словами, если `flizny` есть массив, то приведенное ниже выражение будет истинным:

```
flizny == &flizny[0];  // именем массива является адрес первого элемента
```

Оба имени, `flizny` и `&flizny[0]` представляют адрес в памяти первого элемента массива. (Вспомните, что `&` — это операция адресации.) Оба имени являются константами, поскольку остаются фиксированными в течение всего времени действия программы. В то же время они могут быть присвоены в виде значений *переменной* типа указатель, и вы можете менять значение переменной, как показано в листинге 10.8. Обратите внимание на то, что происходит со значением указателя, когда вы прибавляете к нему число. (Как говорилось ранее, спецификатор `%p`, предназначенный для указателей, обычно отображает шестнадцатеричные значения.)

Листинг 10.8. Программа `pnt_add.c`

```
// pnt_add.c -- сложение указателей
#include <stdio.h>
#define SIZE 4

int main(void)
{
   short dates [SIZE];
   short * pti;
   short index;
   double bills[SIZE];
   double * ptf;

   pti = dates;              // назначение указателю адреса массива
   ptf = bills;
   printf("%23s %10s\n", "short", "double");
   for (index = 0; index < SIZE; index ++)
      printf("указатели + %d: %10p %10p\n",
              index, pti + index, ptf + index);

   return 0;
}
```

Ниже показаны выходные данные этой программы:

```
                    short     double
указатели + 0: 0x0064fd20 0x0064fd28
указатели + 1: 0x0064fd22 0x0064fd30
указатели + 2: 0x0064fd24 0x0064fd38
указатели + 3: 0x0064fd26 0x0064fd40
```

Во второй строке печатаются начальные адреса двух массивов, в следующей строке показан результат от прибавления к адресу 1 и так далее. Не забывайте, что адреса представлены в шестнадцатеричной форме, поэтому 30 на 1 больше, чем 2f, и на 8 больше, чем 28. Почему так?

```
0x0064fd20 + 1 дает в результате 0x0064fd22?
0x0064fd30 + 1 дает в результате 0x0064fd38?
```

Вам все еще это не понятно? Объяснение достаточно простое! В нашей системе реализована побайтная адресация памяти, в то время как тип `short` использует 2 байта, а тип `double` — 8 байтов. Происходит вот что: когда вы говорите "прибавить 1 к указателю", C добавляет одну *единицу хранения*. В случае массивов это означает, что адрес увеличивается до адреса следующего *элемента*, но не до следующего байта (рис. 10.3). Это одна из причин того, почему вы должны объявлять вид объекта, на который указывает указатель. Адреса недостаточно, поскольку компьютер должен знать, сколько нужно байтов для хранения объекта. (Это справедливо и в отношении указателей на скалярные переменные, в противном случае операция `*pt`, выбирающая значение, не будет работать правильно.)

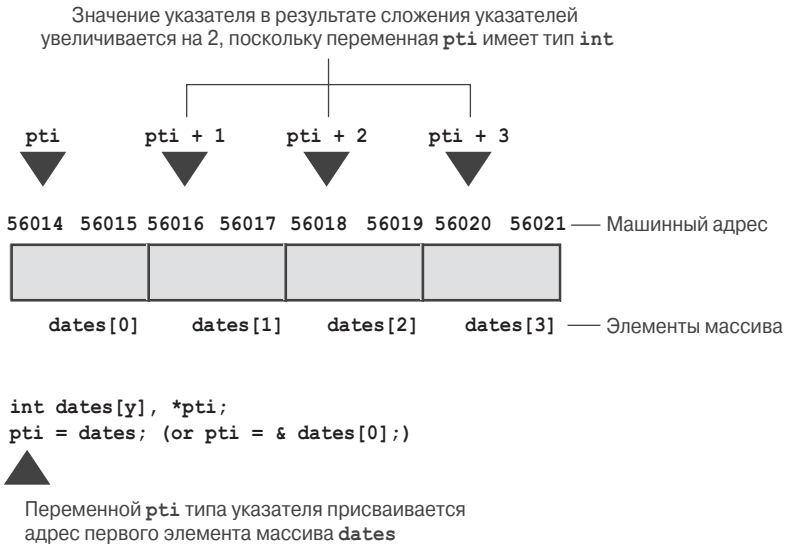

Рис. 10.3. *Массивы и сложение указателей*

Сейчас необходимо дать более четкое определение, что означают выражения "указатель на `int`", "указатель на `float`" или "указатель на любой тип".

- Значение указателя — это адрес объекта, на который он указывает. То, как адрес представлен в памяти машины, зависит от конструктивных особенностей аппаратных средств. Многие компьютеры, в том числе и персональные компьютеры IBM PC и Macintosh, имеют *байтовую адресацию*, это значит, что байты памяти пронумерованы последовательно. В подобных случаях адрес крупного объекта, такого как переменная типа `double`, как правило, представляет собой адрес первого байта объекта.
- Применяя операцию `*` к указателю, получаем значение, которое хранится в объекте, на который нацелен указатель.
- Добавление к указателю 1 увеличивает его значение на величину размера типа в байтах переменной, на которую он указывает.

Благодаря интеллектуальному характеру языка C, имеем следующие равенства:

```
dates +2 == &date[2]       /* тот же адрес   */
*(dates + 2) == dates[2]   /* то же значение */
```

Представленные отношения подводят итог тесной зависимости между массивами и указателями. Это означает, что вы можете использовать указатели для идентификации конкретного элемента массива и получения его значения. По сути, в вашем распоряжении имеются два различных обозначения одного и того же объекта. В самом деле, стандарт языка C описывает массивы через указатели. То есть, он определяет, что `ar[n]` означает `*(ar + n)`. Вы можете интерпретировать второе выражение как "перейти к ячейке памяти `ar`, пройти через `n` единиц и там найти нужное значение".

В то же время, не путайте `*(dates+2)` с `*dates+2`. Операция разыменования (`*`) имеет более высокий приоритет, чем `+`, следовательно, последнее выражение означает `(*dates)+2`:

```
*(dates +2)      /* значение третьего элемента массива dates   */
*dates +2        /* 2 прибавляется к значению первого элемента */
```

Такая зависимость между массивами и указателями означает, что вы часто можете пользоваться любым из подходов при написании программы. Программа в листинге 10.9, например, после компиляции генерирует те же выходные данные, что и программа в листинге 10.1.

Листинг 10.9. Программа `day_mon3.c`

```
/* day_mon3.c -- используются обозначения через указатели */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
   int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
   int index;

   for (index = 0; index < MONTHS; index++)
      printf("Месяц %2d имеет %d дней (ltym).\n", index +1,
              *(days + index));       // то же, что и days[index]
   return 0;
}
```

В рассматриваемом случае `days` представляет собой адрес первого элемента массива, индекс `days + index` — адрес элемента `days[index]`, а `*(days + index)` — значение этого элемента, так же как и `days[index]`. Цикл поочередно просматривает элементы массива и выводит на печать значения, которые в них находит. Обладает ли какими-либо преимуществами программа, написанная подобным образом? По сути, никаких преимуществ она не предоставляет. Основное назначение программы в листинге 10.9 состоит в том, чтобы показать, что подходы с использованием записи через массивы и через указатели являются эквивалентными. Этот пример демонстрирует, что вы можете использовать для массивов систему обозначения с помощью указателей. Обратное утверждение также верно — вы можете использовать систему обозначения массивов в отношении указателей. Это имеет значение в случае работы с функцией, аргументом которой является массив.

Функции, массивы и указатели

Предположим, что вы хотите написать функцию, которая выполняет операции над массивами. Например, нужна функция, возвращающая сумму элементов массива. Предположим, что `marbles` — имя массива значений типа `int`. Какой вид будет иметь вызов такой функции? Здравый смысл подсказывает, что он должен иметь вид:

```
total = sum(marbles);            // возможный вызов функции
```

Каким должен быть прототип этой функции? Вспомните, что именем массива является адрес его первого элемента, так что фактический аргумент `marbles`, будучи адресом значения `int`, должен быть присвоен формальному параметру, каковым является указатель на тип `int`:

```
int sum(int * ar);               // соответствующий прототип
```

Какую информацию получает функция `sum()` благодаря этому аргументу? Она получает адрес первого элемента массива, она также узнает, как найти значение `int` в этой ячейке. Обратите внимание на то, что эта информация ничего не говорит о количестве элементов в массиве. Мы поставлены перед выбором одного из двух вариантов продолжения определения функции. Первый вариант заключается в том, чтобы каким-то образом закодировать фиксированный размер массива в функцию:

```
int sum(int *ar)                 // соответствующее определение
{
   int i;
   int total = 0;
   for( i = 0; i < 10; i++)      // предполагается наличие 10 элементов
        total += ar[i];          // ar[i] то же, что и *(ar + i)
   return total;
}
```

В данном случае используется тот факт, что аналогично возможности применения системы обозначений через указатели к массивам, можно употреблять систему обозначений массивов к собственно указателям. Кроме того, вспомните, что операция `+=` добавляет значение операнда, стоящего справа от знака операции, к операнду, стоящему слева от знака операции. Следовательно, итоговое значение представляет собой сумму элементов массива.

Определение этой функции ограничено; она может работать только с массивами, содержащими 10 элементов. Более гибкий подход состоит в том, что размер массива передается в качестве второго аргумента:

```
int sum(int * ar, int n)         // более общий подход
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)       // используются n элементов
      total += ar[i];            // ar[i] - это то же, что и *(ar + i)
   return total;
}
```

В данном случае первый параметр говорит функции о том, где можно найти массив, и какой тип данных элементов массива, а второй параметр уведомляет функцию о том, сколько элементов содержится в массиве.

О параметрах функции необходимо сказать следующее. В контексте прототипа функции или заголовка определения функции, и *только* в этом контексте, можно поставить `int ar[]` вместо `int * ar`:

```
int sum (int ar[], int n);
```

Форма `int * ar` всегда означает, что `ar` является типом указателя на значение `int`. Форма `int ar[]` также означает, то `ar` — это тип указатель на `int`, однако только в тех случаях, когда он используется *только* для объявления формальных параметров. Идея заключается в том, что вторая форма напоминает, что `ar` не просто указывает на значение типа `int`, он указывает на значение `int`, которое является элементом массива.

Объявление параметров массива

Поскольку имя массива — это адрес его первого элемента, фактический аргумент в виде имени массива требует, чтобы соответствующий формальный аргумент был указателем. В этом контексте, и только в нем, C интерпретирует `int ar[]` как `int * ar` , то есть `ar` является указателем на тип `int`. Поскольку прототипы позволяют опускать имя, все четыре показанных ниже прототипа эквивалентны:

```
int sum(int *ar, int n);
int sum(int *, int);
int sum(int ar[], int n);
int sum(int [], int);
```

Вы не можете опускать имена в определениях функций, следовательно, с точки зрения определений, следующие две формы эквивалентны:

```
int sum(int *ar, int n)
{
   // здесь находится программный код
}
int sum(int ar[], int n);
{
   // здесь находится программный код
}
```

Вы должны иметь возможность использовать любой из указанных выше прототипов с любым из двух определений, приведенных выше.

В листинге 10.10 представлена программа, использующая функцию `sum()`. Чтобы продемонстрировать интересные особенности аргументов типа массива, программа распечатывает также размер исходного файла и размер параметра функции, представляющего массив. (Используйте спецификатор `%u` или, возможно, `%lu`, если ваш компилятор не поддерживает спецификатор `%zd` для печати значений функции `sizeof`.)

Листинг 10.10. Программа `sum_arr1.c`

```
// sum_arr1.c -- сумма элементов массива
// используйте спецификаторы %u или %lu, если спецификатор %zd не работает
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);
```

```
int main(void)
{
   int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
   long answer;

   answer = sum(marbles, SIZE);
   printf("Общая сумма элементов массива marbles равна %ld.\n", answer);
   printf("Объем памяти, отведенной под массив marbles, составляет %zd байт.\n",
            sizeof marbles);
   return 0;
}

int sum(int ar[], int n)          // каков размер массива?
{
   int i;
   int total = 0;

   for( i = 0; i < n; i++)
        total += ar[i];
   printf("Размер переменной ar составляет %zd байт.\n", sizeof ar);

   return total;
}
```

Выходные данные нашей системы имеют следующий вид:

```
Размер переменной ar составляет 4 байт.
Общая сумма элементов массива marbles равна 190.
Объем памяти, отведенной под массив marbles, составляет 40 байт.
```

Обратите внимание на то, что размер массива `marbles` равен 40 байтов. Это имеет смысл, поскольку массив `marbles` содержит 10 значений типа `int`, каждое из которых занимает 4 байта, что в сумме составляет 40 байт. В то же время размер `ar` равен всего 4 байта. Это объясняется тем, что `ar` не является массивом, это указатель на первый элемент массива `marbles`. Наша система использует четырехбайтовый адрес, таким образом, размер переменной типа указатель составляет 4 байта. (Другие системы могут использовать для этой цели другое число байтов.) Короче говоря, в программе из листинга 10.10 `marbles` — это массив, `ar` — указатель на первый элемент массива `marbles`, а связь в C между массивами и указателями позволяет использовать систему обозначений массива для указателя `ar`.

Использование параметров типа указатель

Функция, работающая с массивом, должна знать, где начинать и где заканчивать свои действия. Функция `sum()` использует параметр типа указатель с тем, чтобы распознать начало массива и целочисленный параметр, задающий количество элементов массива, подлежащих обработке. (Параметр типа указатель также описывает тип данных в массиве.) Но это не единственный путь сообщить функции то, что она должна знать. Другой способ состоит в том, чтобы описать массив, передавая функции два параметра, при этом первый из них указывает, где начинается массив (как и раньше), а второй — где массив заканчивается. Программа в листинге 10.11 служит иллюстрацией этого подхода. Он также использует тот факт, что параметр типа указатель является переменной, а это означает, что вместо использования индекса для указания, к какому

элементу массива осуществлять доступ, эта функция может менять само значение указателя, заставляя его поочередно указывать на каждый элемент массива.

Листинг 10.11. Программа `sum_arr2.c`

```
/* sum_arr2.c -- суммирует элементы массива */
#include <stdio.h>
#define SIZE 10
int sump(int * start, int * end);
int main(void)
{
   int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
   long answer;

   answer = sump(marbles, marbles + SIZE);
   printf("Общее количество элементов marbles равно %ld.\n", answer);

   return 0;
}
/* использование арифметики указателей */
int sump(int * start, int * end)
{
   int total = 0;

   while (start < end)
   {
      total += *start;     /* добавить значение к total */
      start++;             /* переместить указатель на следующий элемент */
   }

   return total;
}
```

Указатель `start` в начале указывает на первый элемент массива `marbles`, таким образом, выражение `total +=*start` добавляет значение первого элемента массива (`20`) к значению переменной `total`. Затем выражение `start++` увеличивает значение переменной `start`, благодаря чему она указывает на следующий элемент массива. Поскольку `start` указывает на тип `int`, C увеличивает значение `start` на размер типа `int`.

Обратите внимание на то, что функция `sump()` использует различные методы функции `sum()` для окончания цикла суммирования. Функция `sum()` использует количество элементов в качестве второго аргумента, а в цикле это значение применяется как часть проверки конца цикла:

```
for( i = 0; i < n; i++)
```

Однако функция `sump()` для проверки окончания цикла использует второй указатель:

```
while (start < end)
```

Поскольку выполняется проверка на неравенство, последним элементом массива, подвергнутым обработке, будет элемент, непосредственно предшествующий элементу, на который указывает `end`. Это означает, что `end` указывает на ячейку, которая находится сразу же за последним элементом массива. C гарантирует, что когда он распределяет пространство памяти для массива, указатель на первую ячейку после конца

массива будет допустимым. Благодаря этому обстоятельству, конструкция, подобная данной, также допустима, так как последним значением, которое start получает в цикле, будет end. Обратите внимание на то, что использование указателя, нацеленного "за пределы конца массива", позволяет осуществить такой вызов:

```
answer = sump(marbles, marbles + SIZE);
```

Поскольку индексирование начинается с 0, marbles + SIZE указывает на элемент, следующий за концом массива. Если бы end указывал на последний элемент вместо элемента, следующего за концом массива, вы должны бы были воспользоваться следующим кодом:

```
answer = sump(marbles, marbles + SIZE - 1);
```

Этот код не только менее элегантен по внешнему виду, его, к тому же, труднее запомнить, следовательно, из-за чего в программе появляется тенденция к возникновению ошибок. Между прочим, хотя язык C гарантирует допустимость применения указателя marbles + SIZE, тем не менее, нет таких гарантий в отношении marbles[SIZE], значения, хранимого в этой ячейке.

Вы можете также сжать тело цикла в следующую строку:

```
total += *start++;
```

Унарные операции * и ++ имеют один и тот же приоритет, но они выполняются справа налево. Это означает, что операция ++ применяется к start, но не к *start. Иначе говоря, увеличивается указатель, но не значение, на которое он указывает. Использование постфиксной формы (start++ вместо ++start) означает, что значение показателя не увеличивается, пока значение, на которое нацелен указатель, не будет добавлено к total. Если бы программа использовала конструкцию *++start, порядок был бы следующий: увеличение значения указателя с последующим использованием значения, на которое нацелен указатель. Однако если в программе используется конструкция (*start)++, она использует значение start, после чего увеличивается значение, но не указатель. При этом указатель будет нацелен на тот же элемент, но этот элемент содержит новое число. И хотя запись вида *start++ используется достаточно широко, мы рекомендуем более удобочитаемую форму записи: *(start++). Программа, показанная в листинге 10.12, служит иллюстрацией всех этих "прелестей" приоритетов операций.

Листинг 10.12. Программа order.c

```
/* order.c -- приоритеты операций с указателями */
#include <stdio.h>
int data[2] = {100, 200};
int moredata[2] = {300, 400};
int main(void)
{
   int * p1, * p2, * p3;
   p1 = p2 = data;
   p3 = moredata;
   printf("  *p1 = %d,   *p2 = %d, *   p3 = %d\n",
             *p1     ,   *p2     ,    *p3);
   printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n",
           *p1++ ,     *++p2     , (*p3)++);
```

```
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",
               *p1     ,   *p2     ,     *p3);
    return 0;
}
```

Вот как выглядят выходные данные этой программы:

```
  *p1 = 100,   *p2 = 100,     *p3 = 300
*p1++ = 100, *++p2 = 200, (*p3)++ = 300
  *p1 = 200,   *p2 = 200,     *p3 = 301
```

Единственная операция, которая меняет значение массива — `(*p3)++`. Две другие операции приводят к тому, что указатели `p1` и `p2` перемещаются на следующий элемент массива.

Комментарии: указатели и массивы

Как вы уже убедились, функции, выполняющие обработку массивов, по сути, используют указатели в качестве аргументов, однако при создании функций обработки массивов потребуется выбрать форму записи — с помощью массива или с помощью указателей.

Применение формы записи с использованием массива, как в листинге 10.10, позволяет легче определять, что данная функция работает с массивами. Наряду с этим, запись в форме массива более привычна для программистов, работающих в других языках программирования, таких как, например, FORTRAN, Pascal, Modula-2 или BASIC. Другие программисты, возможно, предпочитают работать с указателями, и для них форма записи с использование указателей, подобная продемонстрированной в листинге 10.11, является более привычной.

Что касается языка C, то два выражения `ar[i]` и `*(ar+i)` эквивалентны по смыслу. Оба работают в тех случаях, когда `ar` есть имя массива, оба они также работают, если `ar` — переменная типа указатель. Тем не менее, использование такого выражения, как `ar++`, работает только в тех случаях, когда `ar` представляет собой переменную типа указатель.

Запись с применением указателей, в частности, когда они сопровождается операцией инкремента, ближе к машинному языку, а при использовании некоторых компиляторов дает более эффективный программный код. В то же время, многие разделяют мнение о том, что основная задача программиста — соблюсти правильность и ясность программного кода, а оптимизация кода вменяется в обязанности компилятора.

Операции с указателями

Так что же можно делать с указателями? Язык C предлагает множество базовых операций, которые можно выполнять над указателями, а приведенная ниже программа демонстрирует восемь из существующих возможностей. Чтобы показать результаты каждой операции, программа выводит на печать значение указателя (таковым является адрес, на который он указывает), значение, хранящееся по адресу, на который нацелен указатель, а также адрес самого указателя. (Если ваш компилятор не поддерживает спецификатор `%p`, попытайтесь воспользоваться для вывода адресов спецификатором `%u` или, возможно, `%lu`.) В листинге 10.13 показаны восемь базовых операций,

которые можно выполнять над переменными типа указатель. В дополнение к этим операциям вы можете использовать операции отношений при сравнении указателей.

Листинг 10.13. Программа `ptr_ops.c`

```
// ptr_ops.c -- операции над указателями
#include <stdio.h>
int main(void)
{
   int urn[5] = {100,200,300,400,500};
   int * ptr1, * ptr2, *ptr3;
   ptr1 = urn;                  // присваивание указателю адреса
   ptr2 = &urn[2];              // второй экземпляр
                                // разыменование указателя и взятие
                                // адреса указателя
   printf("значение указателя, разыменованный указатель, адрес указателя:\n");
   printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
          ptr1, *ptr1, &ptr1);
                                // сложение указателей
   ptr3 = ptr1 + 4;
   printf("\nсложение значения int с указателем:\n");
   printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n",
          ptr1 + 4, *(ptr1 + 3));
   ptr1++;                      // увеличение значение указателя на 1
   printf("\nзначения после выполнения операции ptr1++:\n");
   printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
          ptr1, *ptr1, &ptr1);
   ptr2--;                      // уменьшение значение указателя на 1
   printf("\nзначения после выполнения операции --ptr2:\n");
   printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n",
          ptr2, *ptr2, &ptr2);
   --ptr1;                      // восстановление исходного значения
   ++ptr2;                      // восстановление исходного значения
   printf("\nвосстановление исходных значений указателей:\n");
   printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
                                // вычитание одного указателя из другого
   printf("\nвычитание одного указателя из другого:\n");
   printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %d\n",
          ptr2, ptr1, ptr2 - ptr1);
                                // вычитание целого значения из указателя
   printf("\nвычитание из указателя значения типа int:\n");
   printf("ptr3 = %p, ptr3 - 2 = %p\n",
          ptr3, ptr3 - 2);
   return 0;
}
```

Ниже показаны выходные данные этой программы:

```
значение указателя, разыменованный указатель, адрес указателя:
ptr1 = 0x0012ff38, *ptr1 =100, &ptr1 = 0x0012ff34

сложение значения int с указателем:
ptr1 + 4 = 0x0012ff48, *(ptr4 + 3) = 400
```

```
значения после выполнения операции ptr1++:
ptr1 = 0x0012ff3c, *ptr1 =200, &ptr1 = 0x0012ff34

значения после выполнения операции --ptr2:
ptr2 = 0x0012ff3c, *ptr2 = 200, &ptr2 = 0x0012ff30

восстановление исходных значений указателей:
ptr1 = 0x0012ff38, ptr2 = 0x0012ff40

вычитание одного указателя из другого:
ptr2 = 0x0012ff40, ptr1 = 0x0012ff38, ptr2 - ptr1 = 2

вычитание из указателя значения типа int:
ptr3 = 0x0012ff48, ptr3 - 2 = 0x0012ff40
```

В представленном ниже перечне описаны базовые операции, которые могут быть выполнены над указателями.

- **Присваивание значений**. Указателю можно присвоить адрес. Обычно это делается путем использования имени массива или операции адресации (`&`). В рассматриваемом примере переменной `ptr1` присваивается адрес начала массива `urn`. Этим адресом оказался номер ячейки памяти `0x0012ff38`. Переменная `ptr2` получает в качестве своего значения адрес третьего, и последнего, элемента `urn[2]`. Обратите внимание, что этот адрес должен быть совместим с типом указателя. Иначе говоря, вы не можете присваивать адрес типа `double` указателю, ссылающемуся на значение типа `int`, по крайней мере, без приведения типов, которое в такой ситуации чревато пагубными последствиями. Это правило было введено стандартом C99.
- **Определение значения (разыменование)**. Операция `*` возвращает значение, хранящееся в ячейке, на которую ссылается указатель. По этой причине первоначальным значением `*ptr1` является `100`, это значение хранится в ячейке `0x0012ff38`.
- **Адрес указателя**. Подобно всем переменным, переменные типа указатель имеют адрес и значение. Операция `&` определяет, где хранится сам указатель. В рассматриваемом примере `ptr1` хранится в ячейке `0x0012ff34`. Содержимое этой ячейки памяти — `0x0012ff38`, что является адресом массива `urn`.
- **Сложение целочисленного значения с указателем**. С помощью операции `+` можно сложить целое число с указателем или указатель с целым числом. В любом из этих случаев целое число умножается на количество байт, представляющее тип данных, на который ссылается указатель, после чего результат добавляется к первоначальному адресу. Результат операции `ptr1 + 4` тот же, что и результат операции `&urn[4]`. Результат сложения не определен, если он выходит за пределы массива, на который ссылается исходный указатель, за исключением ссылки на адрес, следующий за концом массива; в этом случае значение считается допустимым.
- **Инкремент значения указателя**. Увеличение значения указателя на величину элемента массива заставляет указатель ссылаться на следующий элемент массива. По этой причине операция `ptr1++` увеличивает значение указателя `ptr1` на `4` (в нашей системе под каждое значение типа `int` отводится 4 байта), в результа-

те указатель ptr1 ссылается на urn[1] (рис. 10.4). Теперь ptr1 имеет значение 0x0012ff3c (адрес следующего элемента массива), а *ptr1 — значение 200 (значение элемента urn[1]). Обратите внимание, что сам адрес ptr1 не меняется и остается равным 0x0012ff34. В конце концов, переменная не перемещается в памяти по причине того, что она поменяла свое значение!

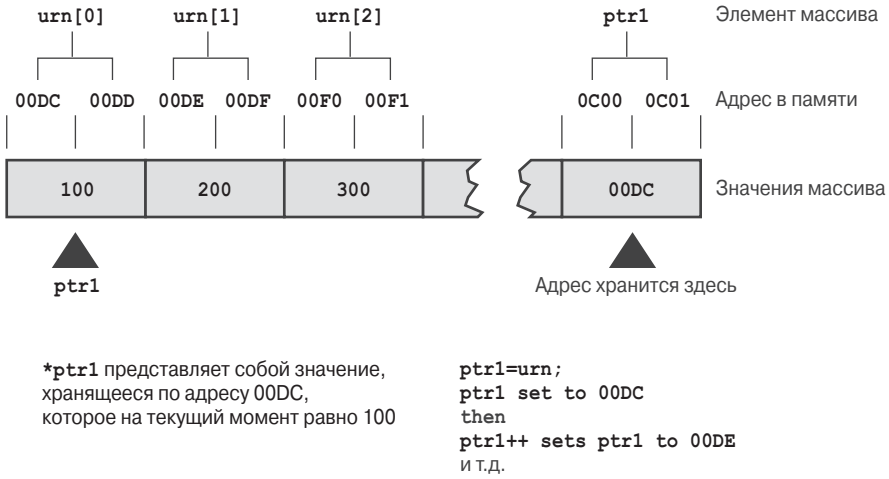

Рис. 10.4. *Увеличение указателя на значение типа* `int`

- **Вычитание целочисленных значений**. С помощью операции - можно вычитать целое число из указателя; указателем должен быть первый операнд или указатель на целое число. Целое число умножается на количество байт в типе, на который ссылается указатель, а результат умножения вычитается из первоначального адреса. Результат операции вычитания ptr3 - 2 аналогичен результату операции &urn[2], поскольку ptr3 указывает на &urn[4]. Результат сложения не определен, если он выходит за пределы массива, на который ссылается исходный указатель, за исключением ссылки на адрес, следующий за концом массива; в этом случае значение считается допустимым.
- **Декремент значения указателя**. Разумеется, значение указателя можно декрементировать. В рассматриваемом примере уменьшение значение ptr2 на единицу приводит к тому, что он ссылается не на третий, а на второй элемент массива. Обратите внимание, что вы можете пользоваться префиксными и постфиксными формами операций инкремента и декремента. Также обратите внимание на то, что оба указателя ptr1 и ptr2 указывали на один и тот же элемент, в данном случае это urn[1] перед тем, как их значение было изменено.
- **Вычисление разности указателей**. Имеется возможность определить разность между двумя указателями. Обычно это делается для двух указателей на элементы, принадлежащие одному и тому же массиву, с тем, чтобы определить, насколько далеко они отстоят друг от друга. Результат представляется в тех же единицах, что и размер типа. Например, в выходных данных программы из листинга 10.13 выражение ptr2 - ptr1 имеет значение 2, что означает, что эти

указатели ссылаются на объекты, разделенные двумя значениями int, но не двумя байтами. Вычитание является гарантировано допустимой операцией, если только оба указателя ссылаются на один и тот же массив (или, возможно, на позицию, следующую непосредственно за концом массива). Применение этой операции к указателям в двух различных массивах может дать какой-то результат, но может привести и к ошибке во время выполнения программы.

- **Сравнения указателей**. Для сравнения значений двух указателей могут использоваться операции отношений в случае, если указатели имеют один и тот же тип.

Обратите внимание на существование двух форм вычитания. Вы можете вычесть один указатель из другого и получить целое число, в то же время вы может вычесть целое число из указателя и получить указатель.

При выполнении операций инкремента и декремента указателя, следует соблюдать определенные меры предосторожности. Компьютер не отслеживает, ссылается ли полученный в результате указатель на какой-то элемент массива. Язык C гарантирует только то, что для заданного массива указатель на любой из элементов этого массива или на позицию, непосредственно следующую за последним элементом массива, является допустимым. Однако результат инкремента или декремента указателя, выходящий за эти пределы, не определен. Наряду с этим, вы можете разыменовать указатель для любого элемента массива. Тем не менее, несмотря на то, что указатель, ссылающийся на элемент, следующий за концом массива, допустим, нет гарантии того, что этот указатель может быть разыменован.

Разыменование неинициализированного указателя

Говоря об осторожности, существует правило, которое должно прочно засесть в вашей памяти: никогда не разыменовывайте неинициализированные указатели. Например, рассмотрим следующий программный код:

```
int * pt;      // неинициализированный указатель
*pt = 5;       // катастрофическая ошибка
```

Почему это настолько плохо? Вторая строка означает необходимость сохранить значение 5 в ячейке, на которую указывает pt. Однако pt, будучи инициализированным, имеет случайное значение, следовательно, неизвестно, куда будет записано значение 5. Оно может не вызвать каких-либо пагубных последствий, но оно может затереть какой-то код или данные, а может вообще привести к аварийному завершению программы. Вспомните, что при создании указателя память выделяется под сам указатель, но для хранения данных память не распределяется. Поэтому, прежде чем вы воспользуетесь указателем, ему должен быть присвоен адрес ячейки памяти, которая уже была распределена. Например, вы можете назначить указателю адрес существующей переменной. (Именно это и происходит, когда вы используете функцию с параметром типа указатель.) Либо вы можете воспользоваться функцией malloc(), которая рассматривается в главе 12, чтобы сначала зарезервировать нужную память.

В любом случае, чтобы не допускать ошибок, не разыменовывайте неинициализированные указатели!

```
double * pd;      // неинициализированный указатель
*pd = 2.4;        // НЕ ДЕЛАЙТЕ этого!
```

Пусть дано

```
int urn[3];
int * ptr1, * ptr2;
```

ниже приведены примеры допустимых и недопустимых операторов:

Допустимый оператор	*Недопустимый оператор*
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

Описанные ранее операции открывают множество возможностей. Программисты, работающие на языке C, создают массивы указателей, указатели на функции, массивы указателей на указатели, массивы указателей на функции и так далее. Однако пусть вас это не волнует, мы ограничимся лишь теми основными видами использования указателей, которые рассматривались выше. Первый основной вид использования указателей состоит в передаче информации из функций и в функции. Вы уже знаете, что вы должны использовать указатели, если хотите, чтобы та или иная функция меняла значения переменных в вызывающей функции. Второй вид использования касается функций, разработанных для манипулирования массивами. Рассмотрим еще один пример программы, использующей функции и массивы.

Защита содержимого массива

Когда вы пишете функцию, которая выполняет обработку данных фундаментального типа, такого как `int`, перед вами встает выбор: передать данные типа `int` по значению или передать указатель на значение типа `int`. Обычное правило предусматривает передачу числовых данных по значению до тех пор, пока в программе не появится потребность изменить это значение, в этом случае вы передаете указатель. Массивы не предоставляют такой возможности, и вы *обязательно* должны передавать указатель. Это делается с целью обеспечить эффективность программы. Если функция передает массив по значению, она должна иметь в своем распоряжении достаточное пространство, чтобы сохранить копию исходного массива, после чего скопировать все данные из исходного массива в новый массив. Гораздо быстрее передать адрес массива, чтобы функция работала с исходным массивом.

При использовании такого метода возникает ряд проблем. Причиной того, что C обычно передает данные по значению, является стремление сохранить целостность данных. Если функция работает с копией исходных данных, она не сможет случайным образом исказить эти данные. В то же время, поскольку функции, выполняющие обработку массива, работают с исходными данными, они *могут* исказить массив. Иногда это желательно. Например, рассмотрим функцию, которая прибавляет одно и то же значение к каждому элементу массива:

```
void add_to(double ar[], int n, double val)
{
   int i;
   for( i = 0; i < n; i++)
      ar[i] += val;
}
```

Следовательно, вызов функции

```
add_to(prices, 100, 2.50);
```

приводит к тому, что каждый элемент массива `prices` получает величину, превосходящую его исходное значение на 2.5; эта функция изменяет содержимое массива. Функция может сделать это, поскольку, работая с указателями, она использует исходные данные.

Другие функции, однако, не должны изменять данные. Следующая функция, например, предназначена для подсчета суммы содержимого массива, и она не меняет массива. Однако, поскольку `ar` есть фактический указатель, программная ошибка может привести к искажению исходных данных. Здесь, например, выражение `ar[i]++` приводит к тому, что значение каждого элемента увеличивается на 1:

```
int sum(int ar[], int n)      // ошибочный программный код
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)
      total += ar[i]++; //по ошибке увеличивается значение каждого элемента
   return total;
}
```

Использование `const` с формальными параметрами

При использовании компилятора K&R C единственный способ избежать ошибок такого вида — быть постоянно бдительным. В стандарте ANSI C существует альтернатива. Если функция не предназначена для того, чтобы менять содержимое массива, используйте ключевое слово `const` при объявлении формального параметра в прототипе и в определении функции. Например, прототип и определение функций `sum()` должны иметь следующий вид:

```
int sum(const int ar[], int n);       /* прототип */
int sum(const int ar[], int n)        /* определение */
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)
      total += ar[i];
   return total;
}
```

Это ключевое слово сообщает компилятору о том, что функция должна рассматривать массив, на который ссылается указатель `ar`, как массив, содержащий константные данные. В этом случае, если вы случайно используете такие выражения, как `ar[i]++`, компилятор может отловить этот случай и выдать сообщение об ошибке, уведомляющее о том, что функция предпринимает попытку изменить константные данные.

Важно понимать, что использование ключевого слова `const` в этом плане не требует, чтобы все элементы исходного массива были константами, оно просто говорит о том, что функция должна обращаться с элементами массива *так, как если бы* они были константами.

Использование ключевого слова const подобным образом обеспечивает защиту массивов, какую передача по значению обеспечивает для фундаментальных типов. В общем случае, если вы пишете функцию, предназначенную для модификации массива, не используйте ключевое слово const при объявлении параметров массива. Если вы пишете функцию, не предназначенную для модификации массива, вы можете указать ключевое слово const при объявлении параметров массива.

В программе, показанной в листинге 10.14, одна функция отображает массив, а другая — умножает каждый элемент массива на заданное значение. Поскольку первая функция не должна менять значения элементов массивов, она использует ключевое слово const. Поскольку вторая функция имеет намерение модифицировать массив, в ней словj const не используется.

Листинг 10.14. Программа `arf.c`

```
/* arf.c -- функции, манипулирующие массивами */
#include <stdio.h>
#define SIZE 5
void show_array(const double ar[], int n);
void mult_array(double ar[], int n, double mult);
int main(void)
{
   double dip[SIZE] = {20.0, 17.66, 8.2, 15.3, 22.22};

   printf("Исходный массив dip:\n");
   show_array(dip, SIZE);
   mult_array(dip, SIZE, 2.5);
   printf("Массив dip после вызова функции mult_array():\n");
   show_array(dip, SIZE);

   return 0;
}

/* выводит содержимое массива */
void show_array(const double ar[], int n)
{
   int i;

   for (i = 0; i < n; i++)
      printf("%8.3f ", ar[i]);
   putchar('\n');
}

/* умножает каждый элемент массива на один и тот же множитель */
void mult_array(double ar[], int n, double mult)
{
   int i;

   for (i = 0; i < n; i++)
      ar[i] *= mult;
}
```

Ниже приводятся выходные данные этой программы:

```
Исходный массив dip:
20.000 17.660 8.200 15.300 22.220
```

```
Массив dip после вызова функции mult_array():
50.000 44.150 20.500 38.250 55.550
```

Обратите внимание, что типом обеих функции является `void`. Функция `mult_array()` присваивает новые значения элементам массива `dip`, однако при этом механизм возврата значений не используется.

Дополнительные сведения о ключевом слове `const`

Как вы убедились выше, ключевое слово `const` можно использовать для создания символических констант:

```
const double PI = 3.14159;
```

В этих случаях, наряду с прочим, вы могли кое-что сделать в отношении директивы `#define`, в то же время ключевое слово `const` дополнительно позволяет создать массивы констант, константные указатели, а также указатели на константы. В листинге 10.4 показано, как используется ключевое слово `const` для защиты массива от модификации:

```
#define MONTHS 12
...
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Если в дальнейшем программный код попытается изменить массив, вы получите сообщение об ошибке на этапе компиляции:

```
days[9] = 44;     /* ошибка на этапе компиляции */
```

Указатели на константы не могут использоваться для изменения значений. Рассмотрим следующий программный код:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * pd = rates;       // pd указывает на начало массива
```

Вторая строка кода объявляет, что значение типа `double`, на которое указывает `pd`, есть `const`. Это означает, что вы не можете использовать `pd` для изменения значений, на которые ссылаются указатели:

```
*pd = 29.89;        // не допускается
pd[2] = 222.22;     // не допускается
rates[0] = 99.99;   // допускается, поскольку rates не является константой
```

Независимо от того, употребляете ли вы форму записи с использованием указателей или с использованием массива, вы не можете использовать `pd` для изменения данных, на которые указывают указатели. Однако, обратите внимание на то, что массив `rates` не был объявлен как константа, вы можете продолжать использовать `rates` с тем, чтобы менять его значения. В то же время, вы можете использовать указатель `pd` для ссылки на какой-то другой объект:

```
pd++;     /* заставляет pd указывать на rates[1] - допустимо */
```

Указатель на константу обычно применяется в качестве параметра функции, чтобы указать, что функция не использует указателей для изменения данных. Например, для функции `show_array()` из листинга 10.14 можно предусмотреть следующий прототип:

```
void show_array(const double *ar, int n);
```

Существует несколько правил, касающихся назначения указателей и употребления ключевого слова const, которые вы должны твердо знать. Во-первых, допускается присваивание адреса как константных, так и неконстантных данных указателю на константу:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
const double * pc = rates;          // допустимо
pc = locked;                        // допустимо
pc = &rates[3];                     // допустимо
```

В то же время обычным указателям могут быть присвоены только адреса неконстантных данных:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
double * pnc = rates;               // допустимо
pnc = locked;                       // не допустимо
pnc = &rates[3];                    // допустимо
```

Это разумное правило. В противном случае вы можете использовать указатель для изменения данных, которые рассматривались как константные.

Последствия применения этих правил на практике состоят в том, что функция, например, show_array(), может принимать имена обычных массивов и постоянных массивов в качестве фактических аргументов, поскольку каждый из них может быть присвоен указателю на константу:

```
show_array(rates, 5);               // допустимо
show_array(locked, 4);              // допустимо
```

В то же время функция, подобная mult_array(), не может принимать имени константного массива в качестве аргумента:

```
mult_array(rates, 5, 1.2);          // допустимо
mult_array(locked, 4, 1.2);         // не допустимо
```

В силу этого обстоятельства использование ключевого слова в определении параметра функции не только обеспечивает защиту данных, он также позволяет функции работать с массивами, которые были объявлены как const.

Существуют и другие варианты использования ключевого слова const. Например, вы можете объявить и инициализировать указатель, который не может указывать на что угодно. Все зависит от того, где размещается ключевое слово const:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
double * const pc = rates;       // pc указывает на начало массива
pc = &rates[2];                  // не допустимо
*pc = 92.99;                     // правильно – изменяет элемент
rates[0]
```

Такой указатель все еще может использоваться для изменения значений, но он может указывать только на ячейку, первоначально присвоенную ему.

И, наконец, вы можете воспользоваться ключевым словом дважды для создания указателя, который не может изменить ни адрес, на который он указывает, ни значение, на которое он указывает:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * const pc = rates;
pc = &rates[2];                 // не допустимо
*pc = 92.99;                    // не допустимо
```

Указатели и многомерные массивы

Как указатели связаны с многомерными массивами? И почему это необходимо знать? Функции, которые работают с многомерными массивами, используют с этой целью указатели, поэтому вы должны продолжить изучение указателей, прежде чем переходить к работе с такими функциями. А что касается первого вопроса, то рассмотрим несколько примеров и постараемся найти на него ответ. Чтобы упростить этот процесс, будем работать с массивами небольших размеров. Предположим, что имеется следующее объявление:

```
int zippo[4][2];           /* массив массивов значений int */
```

В этом случае `zippo`, будучи именем массива, представляет собой адрес первого элемента массива. Тогда первый элемент массива `zippo` сам является массивом, состоящим их двух значений типа `int`, следовательно, `zippo` — это адрес массива, состоящего из двух значений типа `int`. Проведем дальнейший анализ с учетом свойств указателей:

- Поскольку `zippo` — адрес первого элемента массива, `zippo` имеет то же значение, что и `&zippo[0]`. Далее, `zippo[0]` сам по себе есть массив из двух целых чисел, следовательно, `zippo[0]` имеет то же значение, что и `&zippo[0][0]`, адрес его первого элемента, то есть `int`. Короче говоря, `zippo[0]` есть адрес объекта, размер которого выражается через величину типа `int`, а `zippo` — адрес объекта, имеющего размер, равный размеру двух типов `int`. Поскольку и целое значение, и массив, состоящий из двух целочисленных значений, начинаются в одной и той же ячейке, `zippo` и `zippo[0]` имеют одно и то же числовое значение.
- Добавление 1 к указателю или адресу дает значение, которое больше исходного на размер объекта ссылки. В этом отношении `zippo` и `zippo[0]` отличаются друг от друга, поскольку `zippo` ссылается на объект, имеющий размер двух типов `int`, а `zippo[0]` ссылается на объект размером в один тип `int`. Следовательно, `zippo + 1` имеет значение, отличное от `zippo[0] + 1`.
- Разыменование указателя или адреса (применение операции `*` или операции `[]` с индексом) позволяет получить значение, представляемое объектом ссылки. Поскольку `zippo[0]` является адресом его первого элемента, `(zippo[0][0])`, `*(zippo[0])` представляет значение, хранящееся в `zippo[0][0]`, то есть значение типа `int`. Аналогично, `*zippo` представляет значение его первого элемента, то есть `zippo[0]`, в то же время `zippo[0]` сам по себе есть адрес значения типа `int`. Это адрес `&zippo[0][0]`, следовательно, `*zippo` есть `&zippo[0][0]`. Применение оператора разыменования к обоим выражениям, приводит к тому, что `**zippo` эквивалентно `*&zippo[0][0]`, при этом оба эти выражения приводятся к `zippo[0][0]`, представляющему собой значение типа `int`. Короче говоря, `zippo` — это адрес адреса, к которому операция разыменования должна быть применена дважды, чтобы получить обычное значение. Адрес адреса или указатель указателя представляют собой примеры *двойного разыменования*.

Разумеется, возрастание размерности массива увеличивает сложность представления в виде указателей. В этой точке большинство изучающих язык C начинают понимать, почему указатели считаются одним из наиболее трудных аспектов языка. Возможно, вам потребуется еще раз почитать о свойствах указателей, которые описаны выше, и только после этого вернуться к рассмотрению программы в листинге 10.15, где выводятся значения некоторых адресов и содержимое массивов.

Листинг 10.15. Программа `zippo1.c`

```
/* zippo1.c -- информация о массиве zippo */
#include <stdio.h>
int main(void)
{
   int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
   printf("   zippo = %p,    zippo + 1 = %p\n",
            zippo,           zippo + 1);
   printf("zippo[0] = %p, zippo[0] + 1 = %p\n",
           zippo[0],      zippo[0] + 1);
   printf("  *zippo = %p,   *zippo + 1 = %p\n",
            *zippo,        *zippo + 1);
   printf("zippo[0][0] = %d\n", zippo[0][0]);
   printf("  *zippo[0] = %d\n", *zippo[0]);
   printf("    **zippo = %d\n", **zippo);
   printf("      zippo[2][1] = %d\n", zippo[2][1]);
   printf("*(*(zippo+2) + 1) = %d\n", *(*(zippo+2) + 1));
   return 0;
}
```

Ниже показаны результаты выполнения этой программы в одной из систем:

```
   zippo = 0x0064fd38,    zippo + 1 = 0x0064fd40
zippo[0] = 0x0064fd38, zippo[0] + 1 = 0x0064fd3c
  *zippo = 0x0064fd38,   *zippo + 1 = 0x0064fd3c
zippo[0][0] = 2
  *zippo[0] = 2
    **zippo = 2
      zippo[1][2] = 3
*(*(zippo+1) + 2) = 3
```

Другие системы могут отображать различные адресные значения, однако отношения будут такими же, что и описанные в этом разделе. Эти выходные данные показывают, что адрес двумерного массива `zippo` и адрес одномерного массива `zippo[0]` — одни и те же адреса. Каждый из них представляет собой адрес первого элемента соответствующего массива, что в числовом виде то же самое, что и `&zippo[0][0]`.

Тем не менее, здесь имеются определенные различия. В нашей системе тип `int` занимает 4 байта. Как уже говорилось выше, адрес `zippo[0]` указывает на 4-байтный объект. Добавляя 1 к `zippo[0]`, мы должны получить адрес, больший исходного на 4. Имя `zippo` представляет собой адрес массива, состоящего из двух значений типа `int`, таким образом, он идентифицирует 8-байтовый объект данных. По этой причине добавление 1 к `zippo` должно иметь своим результатом адрес, превышающий исходный на 8 байтов, что и происходит на самом деле.

Приведенная выше программа показывает, что `zippo[0]` и `*zippo` идентичны, какими они и должны быть. Далее, она показывает, что имя двумерного массива должно быть разыменовано дважды, чтобы можно было получить доступ к значению, хранящемуся в массиве. Это можно сделать, применив дважды операцию разыменования (`*`) или дважды выполнив операцию квадратных скобок (`[ ]`). (Это можно сделать, если выполнить один раз операцию `*` и один раз применив квадратные скобки `[ ]`, однако, рассматривая все эти возможности, давайте не будем отклоняться от главной цели.)

В частности, обратите внимание, что выражение `zippo[2][1]` в системе записи с использованием указателей эквивалентно выражению `*(*(zippo+2) + 1)` в системе записи с использованием указателей. Вы, должно быть, хотя бы раз в своей жизни пытались нарушить это равенство. Будем строить это выражение постепенно, выполняя указанную ниже последовательность шагов:

`zippo`	← адрес первого элемента, состоящего из двух значений типа `int`.
`zippo+2`	← третий элемент массива, состоящий из двух значений типа `int`.
`*(zippo+2)`	← третий элемент, представляющий собой массив из двух значений, следовательно, это адрес его первого элемента, имеющего значение типа `int`.
`*(zippo+2) + 1`	← адрес второго элемента двухэлементного массива, также имеющего значение типа `int`.
`*(*(zippo+2) + 1)`	← значение второго объекта типа `int` в третьей строке (`zippo[2][1]`).

Цель этой причудливой формы записи с использованием указателей состоит не в том, чтобы продемонстрировать только то, что вы овладели ею и можете использовать ее вместо намного более простого выражения `zippo[2][1]`, но и то, что вы вполне можете пользоваться более простой формой записи в виде массива, чтобы извлечь значение, когда встречается указатель на двумерный массив.

На рис 10.5 показана еще одна точка зрения на зависимость между адресами массива, содержимым массива и указателями.

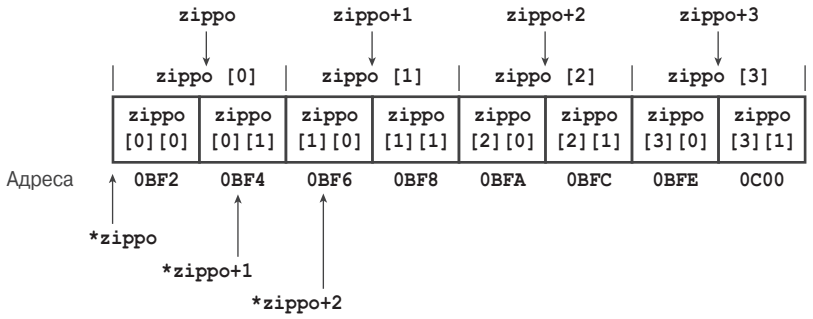

Рис. 10.5. *Массив массивов*

Указатели на многомерные массивы

Как вы объявите переменную pz типа указатель, которая может ссылаться на такой двумерный массив как zippo? Такой указатель может использоваться, например, при написании функции, которая выполняет обработку массивов, подобных zippo. Достаточно ли для этого указателя на значение типа int? Нет, не достаточно. Этот тип совместим с zippo[0], который указывает на какое-то одно значение типа int. Однако zippo есть адрес его первого элемента, который сам представляет собой массив из двух значений типа int. Отсюда следует, что pz должен указывать на массив, содержащий два элемента типа int, а не на какое-то одно значение типа int. Вот как можно поступить в таком случае:

```
int (* pz)[2];     // pz указывает на массив из 2 значений типа int
```

Этот оператор говорит, что pz является указателем на массив из двух значений типа int. Зачем в этом случае нужны круглые скобки? Скобки [ ] имеют более высокий приоритет, чем *. Следовательно, в случае, например, такого объявления

```
int * pax[2];
```

сначала используются квадратные скобки, благодаря которым pax рассматривается как массив двух каких-то объектов данных. Далее, применяется операция *, в результате которой массив pax превращается в массив из двух указателей. В заключение используется значение int, превращающее pax в массив из двух указателей на значения типа int. Это объявление создает *два* указателя на значения типа int, тем не менее, первоначальная версия использует круглые скобки с тем, чтобы сначала применить операцию *, создавая *один* указатель на два значения типа int. Код в листинге 10.16 показывает, как можно использовать такой указатель в качестве исходного массива.

Листинг 10.16. Программа zippo2.c

```
/* zippo2.c -- получение информации о массиве zippo с помощью
               переменной типа указатель */
#include <stdio.h>
int main(void)
{
   int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
   int (*pz)[2];
   pz = zippo;
   printf("   pz = %p,    pz + 1 = %p\n",
              pz,         pz + 1);
   printf("pz[0] = %p, pz[0] + 1 = %p\n",
           pz[0],      pz[0] + 1);
   printf("  *pz = %p,   *pz + 1 = %p\n",
             *pz,        *pz + 1);
   printf("pz[0][0] = %d\n", pz[0][0]);
   printf("  *pz[0] = %d\n", *pz[0]);
   printf("    **pz = %d\n", **pz);
   printf("      pz[2][1] = %d\n", pz[2][1]);
   printf("*(*(pz+2) + 1) = %d\n", *(*(pz+2) + 1));
   return 0;
}
```

Ниже представлены новые выходные данные:

```
pz = 0x0064fd38,    pz + 1 = 0x0064fd40
pz[0] = 0x0064fd38, pz[0] + 1 = 0x0064fd3c
*pz = 0x0064fd38,     *pz + 1 = 0x0064fd3c
pz[0][0] = 2
  *pz[0] = 2
    **pz = 2
      pz[2][1] = 3
*(*(pz+2) + 1) = 3
```

И снова вы можете получить различные адреса, однако отношения остаются прежними. Как мы обещали, вы можете использовать такую форму записи, как `pz[2][1]`, даже если `pz` является указателем, но не именем массива. В более общих случаях вы можете представлять отдельные элементы с помощью формы записи с использованием массива или с использованием указателей применительно либо к имени массива, либо к указателю:

```
zippo[m][n] == *(*(zippo + m) + n)
pz[m][n] == *(*(pz + m) + n)
```

Совместимость указателей

Правила присвоения одного указателя другому обладают более высоким приоритетом, чем правила для числовых значений. Например, вы можете присвоить значение типа `int` переменной типа `double` без применения преобразования типов, в то же время вы не можете поступить подобным образом в отношении указателей на два эти типа:

```
int n = 5;
double x;
int * p1 = &n;
double * pd = &x;
x = n;              // неявное преобразование типа
pd = p1;            // ошибка этапа компиляции
```

Эти ограничения распространяются на более сложные типы. Предположим, что мы имеем следующие объявления:

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2;           // указатель на указатель
```

Тогда мы получаем следующие равенства:

```
pt = &ar1[0][0];   // оба - указатели на значение типа int
pt = ar1[0];       // оба - указатели на значение типа int
pt = ar1;          // не допустимо
pa = ar1;          // оба - указатели на int[3]
pa = ar2;          // не допустимо
p2 = &pt;          // оба - указатели на значение типа int *
*p2 = ar2[0];      // оба - указатели на значение типа int
p2 = ar2;          // не допустимо
```

Обратите внимание, что для всех недопустимых случаев присваивания характерно прежде всего то, что используются два указателя, которые указывают на разные типы данных. Например, pt указывает на одно значение int, в то же время ar1 — на массив из трех значений типа int. Аналогично, pa указывает на массив из двух значений типа int, следовательно, он совместим с ar1, но не с ar2, который указывает на массив из двух значений int.

Два последних примера несколько искусственны. Переменная p2 представляет собой указатель на указатель на значение типа int, в то время как ar2 есть указатель на массив, состоящий из двух значений типа int (или, короче, указатель на массив int[2]). Таким образом, p2 и ar2 — разные типы, и вы не можете присвоить ar2 указателю p2. В то же время *p2 имеет тип указателя на тип int, что делает его совместимым с ar2[0]. Напомним, что ar2[0] является указателем на его первый элемент, в данном случае, ar2[0][0], что также делает ar2[0] типом указателя на значение int.

В общем случае многократные операции разыменования также носят искусственный характер. Например, рассмотрим следующий фрагмент кода:

```
int * p1;
const int * p2;
const int ** pp2;
p1 = p2;          // недопустимо – присваивание const значению не const
p2 = p1;          // допустимо - присваивание не const значению const
pp2 = &p1;        // недопустимо - присваивание не const значению const
```

Как вы убедились выше, присваивание указателя типа const указателю типа не const не допускается, поскольку вы можете использовать новый указатель для изменения данных типа const. В то же время указатель типа не const на указатель const допустим при условии, что вы выполняете всего лишь один уровень разыменования:

```
p2 = p1;         // допустимо - присваивание не const значению const
```

Однако такие присваивания теперь чреваты ошибками, если вы перейдете на два уровня операции разыменования. Если бы они были допустимы, вы бы получили возможность выполнить одну из следующих операций:

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;   // недопустимо, но предполагается допустимость
*pp2 = &n;   // допустимо, оба const, однако устанавливает p1 ссылку на n
*p1 = 10;    // допустимо, но при этом изменяется const
```

Функции и многомерные массивы

Если вы хотите написать функцию, которая выполняет обработку двумерных массивов, то должны иметь достаточно четкое представление об указателях, чтобы делать правильные объявления, касающиеся аргументов функции. В теле самой функции вполне достаточно применять форму записи с помощью массивов.

Напишем функцию, манипулирующую двумерными массивами. Одной из возможностей является использование цикла for для применения функций обработки одномерного массива к каждой строке двумерного массива.

Другими словами, вы можете делать нечто подобное:

```
int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };
int i, j;
int total = 0;
for (i = 0; i < 3 ; i++)
   total += sum(junk[i], 4);   // junk[i] - одномерный массив
```

Вспомните, что если `junk` есть двумерный массив, то `junk[i]` — одномерный массив, который вы можете рассматривать как строку двумерного массива. В этом случае функция `sum()` вычисляет промежуточную сумму для каждой строки двумерного массива, а в цикле `for` выполняется суммирование этих промежуточных сумм.

Однако в условиях этого подхода теряется связь с информацией, касающейся строк и столбцов. В этом приложении (суммирование всех значений) такая информация не имеет решающего значения, однако предположим, что каждая строка представляет год, а каждая строка — месяц. В этом случае вам, возможно, понадобится функция, которая, скажем, суммирует значения, содержащиеся в некотором конкретном столбце. В этом случае функция должна иметь в своем распоряжении информацию о столбцах и строках. Этот вопрос можно решить путем объявления правильного вида формальной переменной, благодаря чему функция могла бы правильно передавать массивы. В этом случае массив `junk` представляет собой массив трех массивов, каждый из которых содержит четыре значения типа `int`. Как показывают ранее проведенные обсуждения, массив `junk` — это указатель на массив из четырех значений типа `int`. Вы можете объявить параметр функции этого типа следующим образом:

```
void somefunction( int (* pt)[4] );
```

С другой стороны, если (и только если) `pt` является формальным параметром функции, вы можете объявить его следующим образом:

```
void somefunction( int pt[][4] );
```

Обратите внимание на то, что первый набор квадратных скобок пуст. Пустые квадратные скобки показывают, что `pt` является указателем. Такого рода переменная может быть использована в этом случае так же, как и массив `junk`. Вот что мы сделали в следующем примере, представленном на листинге 10.17. В этом листинге представлены три эквивалентных формы синтаксиса прототипов.

Листинг 10.17. Программа `array2d.c`

```
// array2d.c -- функции для двумерных массивов
#include <stdio.h>
#define ROWS 3
#define COLS 4
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int );      // можно опустить имена
int sum2d(int (*ar)[COLS], int rows);   // другой вид синтаксиса
int main(void)
{
   int junk[ROWS][COLS] = {
           {2,4,6,8},
           {3,5,7,9},
           {12,10,8,6}
   };
```

```
   sum_rows(junk, ROWS);
   sum_cols(junk, ROWS);
   printf("Сумма всех элементов = %d\n", sum2d(junk, ROWS));
   return 0;
}
void sum_rows(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot;
   for (r = 0; r < rows; r++)
   {
      tot = 0;
      for (c = 0; c < COLS; c++)
          tot += ar[r][c];
      printf("строка %d: сумма = %d\n", r, tot);
   }
}
void sum_cols(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot;
   for (c = 0; c < COLS; c++)
   {
       tot = 0;
       for (r = 0; r < rows; r++)
           tot += ar[r][c];
       printf("столбец %d: сумма = %d\n", c, tot);
   }
}
int sum2d(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
       for (c = 0; c < COLS; c++)
           tot += ar[r][c];
   return tot;
}
```

Ниже показаны выходные данные этой программы:

```
строка 0: сумма = 20
строка 1: сумма = 24
строка 2: сумма = 36
столбец 0: сумма = 17
столбец 1: сумма = 19
столбец 2: сумма = 21
столбец 3: сумма = 23
Сумма всех элементов = 80
```

Программа из листинга 10.17 передает в качестве аргумента имя junk, которое представляет указатель на первый элемент, каковым является подмассив, а также символическую константу ROWS со значением 3, то есть количество строк массива. После этого каждая функция рассматривает ar как массив массивов, содержащих четыре значения типа int. Количество столбцов встраивается в функцию, но количество строк остается неуказанным. Эта же функция будет работать, скажем, с массивом размерности 12×4, если передается число 12 в качестве числа строк. Это объясняется тем, что rows — это количество элементов; в то же время, поскольку каждый элемент представляет собой массив, или строка, rows становится количеством строк.

Обратите также внимание и на то, что ar используется так же, как junk в функции main(). Это становится возможным, поскольку ar и junk имеют один и тот же тип: указатель на массив из четырех значений типа int.

Имейте в виду, что следующее далее объявление не будет работать должным образом:

```
int sum2(int ar[][], int rows);       // ошибочное объявление
```

Вспомните, что компилятор переводит форму записи в стиле массива, в форму записи в стиле указателя. Это значит, например, что ar[1] преобразуется в ar+1. Чтобы компилятор мог вычислить эти выражения, он должен знать размер объекта, на который указывает ar. Объявление

```
int sum2(int ar[][4], int rows);      // правильное объявление
```

говорит о том, что ar указывает на массив, состоящий из четырех значений типа int (в нашей системе, на объект длиной 16 байтов), следовательно, ar+1 означает "прибавить 16 байтов к адресу". В условиях варианта с пустыми квадратными скобками компилятор не будет знать, что делать дальше.

Вы можете также заключить размер в другую пару квадратных скобок, как показано выше, однако компилятор их проигнорирует:

```
int sum2(int ar[3][4], int rows);     // допустимое объявление,
                                      // однако 3 игнорируется
```

Это удобно в тех случаях, когда используются typedef:

```
typedef int arr4[4];       // массив arr4, состоящий из 4 значений int
typedef arr4 arr3x4[3];    // массив arr3x4, состоящий из 3 массивов arr4
int sum2(arr3x4 ar, int rows);    // то же, что и следующее объявление
int sum2(int ar[3][4], int rows); // то же, что и следующее объявление
int sum2(int ar[][4], int rows);  // стандартная форма
```

В общем случае, чтобы объявить указатель, соответствующий *N*-мерному массиву, вы должны снабдить значениями все комплекты квадратных скобок, кроме самой левой пары:

```
int sum4d(int ar[][12][20][30], int rows);
```

Это объясняется тем, что первый комплект квадратных скобок указывает на наличие указателя, в то время как остальные квадратные скобки описывают типы объектов данных, на которые ссылаются указатели, как показано в следующем эквиваленте прототипа:

```
int sum4d(int (*ar)[12][20][30], int rows);   // ar - это указатель
```

В данном случае ar представляет собой указатель на массив 12×20×30 значений типа int.

Массивы переменной длины

Вы, должно быть, уже заметили некоторую странность, характерную для функций, выполняющих обработку двумерных массивов: вы можете описать количество строк посредством параметра функции, однако количество столбцов встроено в функцию. Например, посмотрите на это определение:

```
#define COLS 4
int sum2d(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
      for (c = 0; c < COLS; c++)
          tot += ar[r][c];
   return tot;
}
```

Далее, предположим, что были объявлены следующие массивы:

```
int array1[5][4];
int array2[100][4];
int array3[2][4];
```

Вы можете использовать функцию `sum2d()` для обработки следующих массивов:

```
tot = sum2d(array1, 5);      // сумма элементов массива 5 x 4
tot = sum2d(array2, 100);    // сумма элементов массива 100 x 4
tot = sum2d(array3, 2);      // сумма элементов массива 2 x 4
```

Это объясняется тем, что количество строк передается в параметр `rows`, представляющий собой переменную. Однако если вы захотите просуммировать массив размерности 6×5, вам придется воспользоваться другой функцией, той, в которой `COLS` принимает значение `5`. Такое поведение есть результатом того факта, что для определения размерности массива вы должны воспользоваться константами; в силу этого обстоятельства, вы должны заменить параметр `COLS` переменной.

Если вы на самом деле хотите создать отдельную функцию, способную работать с любым двумерным массивом, вы сможете сделать это, но при этом получите очень неуклюжий результат. (Вам придется передать массив в виде одномерного массива и иметь под рукой функцию, чтобы вычислять, с какого места начинается каждая строка.) Более того, этот метод недостаточно гладко вписывается в подпрограммы на языке FORTRAN, который позволяет программисту задавать в вызове функции оба размера массива. FORTRAN можно считать древним языком программирования, но в течение многих десятилетий эксперты в области численных методов разработали множество полезных вычислительных библиотек на языке FORTRAN. Ожидалось, что C станет наследником FORTRAN, таким образом, возможность корректного переноса библиотек FORTRAN на C представляется весьма полезной.

Эта потребность для стандарта C99 была главным импульсом, побудившим введение концепции массивов переменной длины, которые позволяют использовать переменные для указания размеров массива.

Например, вы можете сделать следующее:

```
int quarters = 4;
int regions = 5;
double sales[regions][quarters];          // массив переменной длины
```

Как говорилось выше, на использование массивов переменной длины накладываются определенные ограничения. Для них должен быть предусмотрен класс автоматической памяти, это означает, что они объявляются либо в функции, либо как параметры функции. Кроме того, вы не можете их инициализировать в объявлении.

Массивы переменной длины не меняют размера

Термин *переменный* в отношении массивов переменной длины не означает, что можно менять длину массива после того, как вы его создадите. Будучи созданным, массив переменной длины сохраняет свои размеры. Термин *переменный* означает, что вы можете использовать переменные при описании размерности массива.

Поскольку массивы переменной длины представляют собой новое языковое средство, поддержка его в настоящее время пока не отличается полнотой и стабильностью. Рассмотрим простой пример, демонстрирующий, как следует писать функцию, которая выполняет суммирование содержимого любого двумерного массива значений `int`.

Прежде всего, покажем, как объявлять функцию с аргументом в виде двумерного массива переменной длины:

```
int sum2d(int rows, int cols, int ar[rows][cols]);
  // ar - массив переменной длины
```

Обратите внимание на то, что два первых параметра (строки и столбцы) используются как размерность для объявления параметра массива `ar`. Поскольку в объявлении массива `ar` присутствуют строки и столбцы, они должны быть объявлены до того, как `ar` появится в списке параметров. В силу этого обстоятельства следующий прототип является ошибочным:

```
int sum2d(int ar[rows][cols], int rows, int cols); // неправильный порядок
```

Стандарт C99 утверждает, что вы можете в прототипе опускать имена, но в этом случае вы должны заменить опущенные размерности звездочками:

```
int sum2d(int, int, int ar[*][*]);
  // ar - массив переменной длины, имена опущены
```

Во-вторых, посмотрите, как определять функцию:

```
int sum2d(int rows, int cols, int ar[rows][cols])
{
   int r;
   int c;
   int tot = 0;

   for (r = 0; r < rows; r++)
       for (c = 0; c < cols; c++)
           tot += ar[r][c];

   return tot;
}
```

Без учета заголовка новой функции, единственное отличие от классической версии этой функции на языке C (листинг 10.17) состоит в том, что константа `COLS` была заменена переменной `cols`. Массив переменной длины в заголовке функции является именно тем средством, которое позволяет проводить такие изменения. Кроме того, наличие переменных, которые представляют как количество строк, так и количество столбцов массива, позволяет использовать новую функцию `sum2d()` с любыми размерами двумерного массива значений `int`. Листинг 10.18 служит иллюстрацией этого утверждения. В то же время, для этого требуется компилятор языка C, в котором это новое свойство реализовано. Программа в листинге 10.18 также показывает, что эта функция, построенная на основе свойств массивов переменной длины, может использоваться как с любыми традиционными массивами языка C, так и с массивами переменной длины.

Листинг 10.18. Программа `vararr2d.c`

```
//vararr2d.c -- функции, использующие массивы переменной длины
#include <stdio.h>
#define ROWS 3
#define COLS 4
int sum2d(int rows, int cols, int ar[rows][cols]); int main(void)
{
   int i, j;
   int rs = 3;
   int cs = 10;
   int junk[ROWS][COLS] = {
         {2,4,6,8},
         {3,5,7,9},
         {12,10,8,6}
   };
   int morejunk[ROWS-1][COLS+2] = {
         {20,30,40,50,60,70},
         {5,6,7,8,9,10}
   };
   int varr[rs][cs];                          // массив переменной длины
   for (i = 0; i < rs; i++)
      for (j = 0; j < cs; j++)
         varr[i][j] = i * j + j;
   printf("Массив размерности 3x5\n");
   printf("Сумма всех элементов = %d\n",
          sum2d(ROWS, COLS, junk));
   printf("Массив размерности 2x6 \n");
   printf("Сумма всех элементов = %d\n",
          sum2d(ROWS-1, COLS+2, morejunk));
   printf("Массив переменной длины размерности 3x10\n");
   printf("Сумма всех элементов = %d\n",
          sum2d(rs, cs, varr)); return 0;
}
```

```
// функция с параметром в виде массива переменной длины
int sum2d(int rows, int cols, int ar[rows][cols])
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
       for (c = 0; c < cols; c++)
           tot += ar[r][c];
   return tot;
}
```

Выходные данные этой программы имеют следующий вид:

```
Массив размерности 3x5
Сумма всех элементов = 80
Массив размерности 2x6
Сумма всех элементов = 315
Массив переменной длины размерности 3x10
Сумма всех элементов = 270
```

Следует отметить тот факт, что объявление массива в списке параметров объявления функции на самом деле не приводит к созданию массива. Как и в случае старого синтаксиса, фактическое имя массива переменной длины является указателем. Это значит, что функция с параметром в виде массива переменной фактически работает с данными исходного массива, и в силу этого обстоятельства способна вносить изменения в массив, переданный ей в качестве аргумента. Следующий фрагмент кода показывает, когда необходимо объявлять указатель, а когда — фактический массив:

```
   int thing[10][6];
   twoset(10,6,thing);
   ...
}
void twoset (int n, int m, int ar[n][m])
  // ar представляет собой указатель на массив из m значений типа int
{
   int temp[n][m];  // temp - массив размерности n x m значений типа int
   temp[0][0] = 2;  // установить значение элемента массива temp равным 2
   ar[0][0] = 2;    // установить значение элемента thing[0][0] равным 2
}
```

Когда вызывается функция `twoset()`, как показано, `ar` становится указателем на элемент `thing[0]`, а `temp` создается как массив размерности 10×6. Поскольку как `ar`, так и `thing` являются указателями на `thing[0]`, то `ar[0][0]` получает доступ к той же ячейке памяти, что и `thing[0][0]`.

Массивы переменной длины позволяют также реализовать динамическое распределение памяти. Это означает, что вы можете задавать размеры массива во время выполнения программы. Для обычных массивов выполняется статическое распределение памяти, означающее, что размер массив известен во время компиляции. Именно по этой причине размеры массива, являющиеся константами, должны быть известны компилятору заранее. Вопросы динамического распределения памяти рассматриваются в главе 12.

Составные литералы

Предположим, что вы хотите передать в функцию некоторое значение с помощью параметра типа int; вы можете передать переменную типа int, но вы также можете передать константу типа int, такую как 5. До появления стандарта C99 возможности реализации функции с аргументом в виде массива были другими; вы могли передать массив, однако тогда не было ничего такого, что могло бы служить эквивалентом константы типа массив. Стандарт C99 изменил эту ситуацию, вводя *составные литералы*. Литералы — это константы, которые не являются символическими константами. Например, 5 есть литеральная константа типа int, 81.3 — литеральная константа типа double, 'Y' — литерал типа char, а "elephant" — строковая литеральная константа. Комитет, который разрабатывал стандарт C99, пришел к заключению, что удобнее будет иметь составные литеральные константы, которые могут представлять содержимое массивов и структур.

С точки зрения массивов, составной литерал выглядит как список инициализации массивов, которому предшествует имя типа, заключенное в круглые скобки. Например, ниже показано обычное объявление массива:

```
int diva[2] = {10, 20};
```

Здесь используется составная литеральная константа, которая создает безымянный массив, содержащий те же значения типа int:

```
(int [2]){10, 20}              // составная литеральная константа
```

Обратите внимание, что именем типа есть то, у вас остается, когда из удалите diva из предыдущего объявления, то есть int [2].

Точно так же, как вы опускаете размерность массива при инициализации именованного массива, вы можете опустить его в составной литеральной константе, а компилятор сам подсчитает, сколько имеется элементов в массиве:

```
(int []){50, 20, 90}           // составной литерал с тремя элементами
```

Поскольку эти составные литеральные константы не имеют имени, вы можете создавать их в каком-то одном операторе и использовать в дальнейшем по мере необходимости. С другой стороны, вы как-то должны их использовать в момент их создания. Один из способов состоит в применении указателя для отслеживания местоположения соответствующей ячейки памяти. То есть, вы можете написать нечто подобное показанному ниже:

```
int * pt1;
pt1 = (int [2]) {10, 20};
```

Следует также отметить, что эта литеральная константа идентифицируется как массив значений тип int. Подобно имени массива, она преобразуется в адрес первого элемента, таким образом, она может быть присвоена указателю на значение типа int. Этот указатель вы можете использовать позже. Например, *pt1 в этом случае будет иметь значение 10, а pt1[1] — значение 20.

Другой прием, который становится возможным благодаря составным литеральным константам, заключается в том, что вы можете передать его в качестве фактического аргумента в функцию с соответствующим формальным параметром:

```
int sum(int ar[], int n);
...
int total3;
total3 = sum((int []){4,4,4,5,5,5}, 6);
```

В данном случае первый аргумент представляет собой массив, состоящий из шести элементов типа `int`, который действует как адрес первого элемента, то есть так же, как и имя массива. Этот вид использования, в рамках которого вы передаете информацию функции без необходимости заранее создавать массив, типичен для составных литеральных констант.

Вы можете распространить этот метод на двумерные и многомерные массивы. Вот как в этом случае создается двумерный массив значений типа `int` и сохраняется его адрес:

```
int (*pt2)[4]; //объявление указателя на массив массивов 4 значений типа int
pt2 = (int [2][4]) { {1,2,3,-9}, {4,5,6,-8} };
```

В данном случае типом является `int [2][4]`, то есть массив размерности 2×4 значений типа `int`.

Листинг 10.19 объединяет все эти примеры в одну полную программу.

Листинг 10.19. Программа `flc.c`

```
// flc.c -- странно выглядящие константы
#include <stdio.h>
#define COLS 4
int sum2d(int ar[][COLS], int rows);
int sum(int ar[], int n);

int main(void)
{
   int total1, total2, total3;
   int * pt1;
   int (*pt2)[COLS];

   pt1 = (int [2]) {10, 20};
   pt2 = (int [2][COLS]) { {1,2,3,-9}, {4,5,6,-8} };

   total1 = sum(pt1, 2);
   total2 = sum2d(pt2, 2);
   total3 = sum((int []){4,4,4,5,5,5}, 6);
   printf("total1 = %d\n", total1);
   printf("total2 = %d\n", total2);
   printf("total3 = %d\n", total3);

   return 0;
}

int sum(int ar[], int n)
{
   int i;
   int total = 0;
   for( i = 0; i < n; i++)
       total += ar[i];
   return total;
}
```

```
int sum2d(int ar[][COLS], int rows)
{
   int r;
   int c;
   int tot = 0;
   for (r = 0; r < rows; r++)
       for (c = 0; c < COLS; c++)
           tot += ar[r][c];
   return tot;
}
```

Для этой программы необходим компилятор, в котором реализовано рассматриваемое дополнение к стандарту C99 (на текущий момент очень немногие компиляторы способны делать это). Ниже показаны выходные данные этой программы:

```
total1 = 30
total2 = 4
total3 = 27
```

Ключевые понятия

Когда возникает необходимость хранить множество значений одного и того же типа, наиболее подходящим средством для решения этой задачи является массив. Язык C рассматривает массивы как *производные типы*, поскольку они построены на основе других типов. Другими словами, вы не просто объявляете массив значений типа `int` или типа `float` или какого-либо другого типа. Этот другой тип сам по себе является типом массива, и в конечном итоге получается массив массивов, или двумерный массив.

Иногда бывает полезно предусмотреть функции для обработки массивов, это позволяет повысить модульность программы за счет решения конкретных задач в рамках специализированных функций. Важно понимать, что когда вы используете имя массива в качестве фактического аргумента, вы не передаете этой функции весь массив, вы просто передаете адрес массива (следовательно, соответствующий формальный параметр функции должен быть указателем). Чтобы обработать этот массив, функция должна знать, где хранится массив и сколько элементов содержит этот массив. Адрес массива указывает “где”; данные о том “сколько” должны либо быть встроены в функцию, либо передаваться ей в качестве отдельного аргумента. Второй подход носит более общий характер, что позволяет одной и той же функции работать с массивами различной размерности.

Связь между массивами и указателями настолько тесная, что вы часто можете представить одну и ту же операцию, употребляя форму записи с использованием массивов и форму записи через указатели. Именно эта связь позволяет применять форму записи массивов в функции, обрабатывающей массивы даже в тех случаях, когда формальный параметр является указателем, а не массивом.

В языке C вы должны задавать размерность обычного массива константным выражением, благодаря чему размерность обычного массива становится известной во время компиляции. Стандарт C99 предлагает альтернативу в виде массива переменной длины, когда спецификатором размерности может быть переменная. Это позволяет задавать размеры массива переменной длины во время выполнения программы.

Резюме

Массив — это набор элементов одного и того же типа. Элементы хранятся в памяти в виде последовательности, а доступ к ним осуществляется с помощью целочисленного индекса, или *смещения*. В языке C первый элемент массива имеет индекс `0`, следовательно, завершающий элемент массива, содержащего `n` элементов, имеет индекс `n - 1`. Контроль за правильным использованием индексов возлагается на программиста, поскольку ни компилятор, ни исполняемая программа не следят за этим процессом.

Для объявления простого *одномерного* массива используется следующая форма:

```
тип имя[размер];
```

Здесь *тип* указывает тип данных каждого элемента массива, *имя* — это имя массива, а размер задает количество элементов. Традиционно язык C требовал, чтобы размер был константным целочисленным выражением. Стандарт C99 позволяет использовать неконстантное целочисленное выражение; в рассматриваемом случае массив трактуется как массив переменной длины.

Язык C интерпретирует имя массива как адрес первого элемента этого массива. В другой терминологии имя массива эквивалентно указателю на первый элемент массива. В общем случае массивы и указатели тесно связаны друг с другом. Если `ar` есть массив, то выражения `ar[i]` и `*(ar + i)` эквивалентны.

Язык C не допускает передачу всего массива в качестве аргумента функции, однако, вы можете передать в качестве аргумента адрес массива. Далее функция может использовать этот адрес для манипулирования исходным массивом. Если функция не предназначается для модификации исходного массива, вы должны воспользоваться ключевым словом `const` при объявлении формального параметра, представляющего массив. Вы можете использовать в вызывающей функции как запись в форме массива, так и запись в форме указателей. В любом случае на практике используется переменная типа указатель.

Добавление к указателю целого значения или инкремент указателя меняет значение указателя на число байтов, которые занимает в памяти объект, на который нацелен указатель. Иначе говоря, если `pd` указывает на 8-байтовое значение типа `double` в массиве, добавление 1 к указателю `pd` увеличивает его значение на 8, следовательно, этот указатель будет ссылаться на следующий элемент массива.

Двумерные массивы представляют собой массивы массивов. Например, объявление

```
double sales[5][12];
```

создает массив с именем `sales`, состоящий из пяти элементов, каждый из которых представляет собой массив из 12 значений типа `double`.

На первый из этих одномерных массивов можно ссылаться как `sales[0]`, на второй — `sales[1]` и так далее, при этом каждый из этих массивов содержит 12 значений типа `double`. Второй индекс служит для доступа к конкретным элементам в этих массивах. Например, `sales[2][5]` — шестой элемент массива `sales[2]`, а `sales[2]` — третий элемент массива `sales`.

Традиционный метод языка C передачи многомерного массива в функцию предусматривает передачу имени массива, которое является адресом этого массива, параметру в форме указателя на значение подходящего типа.

Объявление такого указателя должно описать все размерности массива, кроме первой; размерность первого параметра обычно передается во втором аргументе. Например, чтобы обработать ранее упоминавшийся массив `sales`, прототип функции и вызов функции должны иметь вид:

```
void display(double ar[][12], int rows);
...
display(sales, 5);
```

Массивы переменной длины позволяют применять второй синтаксис, в рамках которого обе размерности передаются функции в качестве аргументов. В этом случае прототип функции и вызов функции принимают следующий вид:

```
void display(int rows, int cols, double ar[rows][cols]);
...
display(5, 12, sales);
```

В ходе описанных выше рассуждений были использованы массивы значений типа `int` и массивы значений типа `double`, однако все, что было сказано выше, применимо и к массивам других типов. В то же время для символьных строк предусматривается целый набор специальных правил. Это объясняется тем фактом, что завершающий символ пробела в строке позволяет функции обнаружить конец строки без необходимости передачи ей размера. Символьные строки подробно рассматриваются в главе 11.

Вопросы для самоконтроля

1. Что выводит на экран следующая программа?

```
#include <stdio.h>
int main(void)
{
   int ref[] = {8, 4, 0, 2};
   int *ptr;
   int index;
   for (index = 0, ptr = ref; index < 4; index++, ptr++)
      printf("%d %d\n", ref[index], *ptr);
   return 0;
}
```

2. Сколько элементов содержит массив `ref` из вопроса 1?
3. Адресом чего является `ref` в вопросе 1? Что можно сказать о `ref + 1`? На что ссылается выражение `++ref`?
4. Какими являются значения `*ptr` и `*(ptr + 2)` в каждом из следующих случаев?

   а.
```
int *ptr;
int torf[2][2] = {12, 14, 16};
ptr = torf[0];
```

   б.
```
int * ptr;
int fort[2][2] = { {12}, {14,16} };
ptr = fort[0];
```

5. Какие значения принимают выражения `**ptr` и `**(ptr + 1)` в каждом из следующих функций?
   а.
   ```
   int (*ptr)[2];
   int torf[2][2] = {12, 14, 16};
   ptr = torf;
   ```
   б.
   ```
   int (*ptr)[2];
   int fort[2][2] = { {12}, {14,16} };
   ptr = fort;
   ```
6. Предположим, что имеет следующее выражение:
   ```
   int grid[30][100];
   ```
   а. Выразите адрес `grid[22][56]` одним способом.
   б. Выразите адрес `grid[22][0]` двумя способами.
   в. Выразите адрес `grid[0][0]` тремя способами.
7. Напишите соответствующее объявление для каждой из следующих переменных:
   а. `digits` представляет собой массив из 10 значений типа `int`.
   б. `rates` представляет собой массив из шести значений типа `float`.
   в. `mat` представляет собой массив, состоящий из трех массивов, каждый из которых содержит 5 целых значений.
   г. `psa` представляет собой массив, состоящий из 20 указателей, ссылающихся на значение типа `char`.
   д. `pstr` представляет собой указатель на массив, состоящий из 20 значений типа `char`.
8. а. Объявите массив, состоящий из шести значений типа `int`, и инициализируйте его значениями 1, 2, 4, 8, 16 и 32.
   б. Используйте запись в форме массива для представления третьего элемента (имеющего значение 4) массива, заданного в пункте а).
   в. Предполагая, что правила стандарта C99 вступили в силу, объявите массив из 100 значений типа `int` и инициализируйте его таким образом, чтобы последний элемент получил значение `-1`; значения остальных элементов могут быть произвольными.
9. Каким должен быть диапазон значений индексов массива, состоящего из 10 элементов?
10. Предположим, что имеются следующие объявления:
    ```
    float rootbeer[10], things[10][5], *pf, value = 2.2;
    int i = 3;
    ```
    Назовите, какие из приведенных ниже операторов допустимы, а какие — нет:
    а. `rootbeer[2] = value;`
    б. `scanf("%f", &rootbeer );`
    в. `rootbeer = value;`
    г. `printf("%f", rootbeer);`
    д. `things[4][4] = rootbeer[3];`
    е. `things[5] = rootbeer;`
    ж. `pf = value;`
    з. `pf = rootbeer;`

11. Объявите массив значений типа `int` размерности 800×600.
12. Пусть заданы объявления трех массивов:

```
double trots[20];
short clops[10][30];
long shots[5][10][15];
```

    а. Напишите прототип и оператор вызова для обычной функции типа `void`, которая обрабатывает элементы массива `trots`, и для функции, использующей массивы переменной длины.

    б. Напишите прототип и оператор вызова для обычной функции типа `void`, которая обрабатывает элементы массива `clops`, и для функции, использующей массивы переменной длины.

    в. Напишите прототип и оператор вызова для обычной функции типа `void`, которая обрабатывает элементы массива `shots`, и для функции, использующей массивы переменной длины.

13. Заданы два прототипа функций:

```
void show(double ar[], int n);         // n - количество элементов
void show2(double ar2[][3], int n);    // n - количество строк
```

    а. Напишите оператор вызова функции `show()`, который передает функции составную литеральную константу, содержащую значения 8, 3, 9 и 2.

    б. Напишите оператор вызова функции `show()`, который передает функции составную литеральную константу, содержащую 8, 3 и 9 в качестве первой строки и значения 5, 4 и 1 в качестве второй строки.

Упражнения по программированию

1. Внесите изменения в программу `rain`, представленную в листинге 10.7, с таким расчетом, чтобы она выполняла вычисления, используя указатели вместо индексов массивов. (Вам по-прежнему придется объявлять и инициализировать соответствующий массив.)
2. Напишите программу, которая инициализирует некоторый массив значений типа `double`, а затем копирует содержимое этого массива в два других массива. (Все три массива должны быть объявлены в основной программе.) Для создания первой копии воспользуйтесь функцией, в которой применяется запись в форме массива. Для создания второй копии воспользуйтесь функцией, в которой применяется запись в форме указателя и инкрементирование указателя. Каждая функция должна принимать в качестве аргументов имя искомого (целевого) массива и количество элементов, подлежащих копированию. Иначе говоря, с учетом соответствующих объявлений, вызовы функций должны иметь следующий вид:

```
double source[5] = {1.1, 2.2, 3.3., 4.4, 5.5};
double target1[5];
double target2[5];
copy_arr(source, target1, 5);
copy_ptr(source, target1, 5);
```

3. Напишите функцию, которая возвращает наибольшее значение из массива значений типа `int`. Протестируйте эту функцию с помощью простой программы.
4. Напишите функцию, которая возвращает индекс наибольшего значения из массива значений типа `double`. Протестируйте эту функцию с помощью простой программы.
5. Напишите функцию, которая возвращает разность между наибольшим и наименьшим значениями из массива типа `double`. Протестируйте эту функцию с помощью простой программы.
6. Напишите программу, которая инициализирует двумерный массив значений типа `double` и использует одну из функций копирования из упражнения 2 для его копирования во второй двумерный массив. (Поскольку двумерный массив представляет собой массив массивов, функция, предназначенная для копирования одномерных массивов, может использоваться для работы с каждым из подмассивов.)
7. Воспользуйтесь одной из функций копирования из упражнения 2 для копирования элементов с третьего по пятый семиэлементного массива в массив, состоящий из трех элементов. Саму функцию менять не надо, достаточно всего лишь правильно выбрать фактические аргументы. (Фактическими аргументами не обязательно должны быть имя массива и размер массива. Достаточно, чтобы такими аргументами были адрес элемента массива и количество обрабатываемых элементов.)
8. Напишите программу, которая инициализирует двумерный массив значений типа `double` размерности 3×5 и использует функцию, ориентированную на работу с массивами переменной длины, для копирования этого массива во второй двумерный массив. Кроме того, напишите функцию, ориентированную на работу с массивами переменной длины, для отображения содержимого этих двух массивов. В общем случае обе эти функции должны быть способны выполнять обработку произвольных массивов размерности *N*×*M*. (Если вы не имеете доступа к компилятору, способному работать с массивами переменной длины, воспользуйтесь традиционным подходом языка C, применяемым к функциям, которые могут выполнять обработку массивов *N*×5).
9. Напишите функцию, которая устанавливает значение элемента массива равным сумме соответствующих элементов двух других массивов. Иначе говоря, если массив 1 имеет значения `2`, `4`, `5` и `8`, а массив 2 — значения `1`, `0`, `4` и `6`, эта функция присваивает массиву 3 значения `3`, `4`, `9` и `14`. Эта функция должна принимать имена трех массивов и их размерности в качестве аргументов. Протестируйте эту функцию с помощью простой программы.
10. Напишите программу, которая объявляет массив размерностью 3×5 и выполняет его инициализацию значениями по вашему выбору. Программа должна вывести эти значения на экран, удвоить все значения, после чего вывести на экран новые значения. Напишите одну функцию для вывода значений на экран и другую функцию для удваивания значений. В качестве аргументов функции принимают имя массивов и количество строк.

11. Перепишите программу `rain` из листинга 10.7 таким образом, чтобы основные задачи выполнялись соответствующими функциями, но не в теле функции `main()`.
12. Напишите программу, которая предлагает пользователю ввести три набора, каждый из которых содержит по пять чисел типа `double`.
    Программа должна выполнять следующие действия:
    а. Запоминать информацию в массиве размерности 3×5.
    б. Вычислять среднее значение каждого набора из пяти чисел.
    в. Вычислять среднее значение всех чисел.
    г. Определять наибольшее из 15 значений.
    д. Выводить на экран сообщение с результатами вычислений.

    Каждая более-менее крупная задача должна решаться специальной функцией с использованием традиционного для языка C подхода к обработке массивов. Выполните задачу б) с помощью функции, которая вычисляет и возвращает среднее значение одномерного массива; воспользуйтесь циклом для вызова этой функции три раза. Функции, реализующие остальные задачи, должны получать в качестве аргумента весь массив, а функции, выполняющие задачи в) и г) должны возвращать ответ в вызывающую программу.
13. Выполните упражнение 12, но в качестве параметров функции используйте массивы переменной длины.