Массивы и связанные списки

В предыдущих главах переменные (типа `int` и `char`) и объекты (типа `Cat`) объявлялись по одному. Между тем достаточно часто возникает необходимость объявить набор объектов, например, 20 переменных типа `int` или дюжину объектов типа `Cat`.

На сегодняшнем занятии вы узнаете:

- что такое массив и как его объявить;
- что такое строка и как использовать массивы символов для создания строк;
- взаимосвязь между массивами и указателями;
- как использовать арифметические операции над указателями с применением массивов;
- что такое связанный список.

Что такое массив?

Массив — это набор элементов, способных хранить данные одного типа. Каждый элемент хранения называется *элементом массива*.

Объявляя массив, необходимо сначала указать тип хранимых данных, а затем имя массива и его размер. Размером массива называется количество его элементов, указываемое в квадратных скобках.

```
long LongArray[25];
```

Здесь, например, объявлен массив `LongArray` из 25-ти элементов типа `long`. Обнаружив это объявление, компилятор зарезервирует область памяти, достаточную для хранения 25-ти переменных типа `long`. Поскольку каждой переменной типа `long` необходимы четыре байта, весь объявленный набор займет непрерывную область памяти размером 100 байтов, как показано на рис. 13.1.

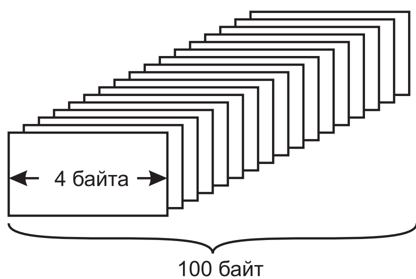

Рис. 13.1. Объявление массива

Доступ к элементам массива

К каждому из элементов можно обратиться по его номеру, расположенному в квадратных скобках после имени массива. Номера элементов массива начинаются с нуля. Следовательно, первым элементом массива `LongArray` будет `LongArray[0]`, вторым — `LongArray[1]` и т.д.

Например, массив `SomeArray[3]` состоит из трех элементов: `SomeArray[0]`, `SomeArray[1]` и `SomeArray[2]`. В общем случае массив `SomeArray[n]`, состоящий из *n* элементов, содержит элементы от `SomeArray[0]` до `SomeArray[n-1]`.

Следовательно, массив `LongArray[25]` пронумерован от `LongArray[0]` до `LongArray[24]`. Листинг 13.1 показывает, как объявить массив из пяти целых чисел и заполнить его значениями.

Начиная с этого момента, номера строк в листингах начинаются с нуля. Это лишний раз напомнит, что массивы в языке C++ начинаются с нуля!

Листинг 13.1. Использование массива целых чисел

```
 0:  // Листинг 13.1. Массивы
 1:  #include <iostream>
 2:
 3:  int main()
 4:  {
 5:    int myArray[5];           // Массив из 5 целых чисел
 6:    int i;
 7:    for (i=0; i<5; i++)       // 0-4
 8:    {
 9:      std::cout << "Value for myArray[" << i << "]: ";
10:      std::cin >> myArray[i];
11:    }
12:    for (i=0; i<5; i++)
13:      std::cout << i << ": " << myArray[i] << endl;
14:    return 0;
15:  }
```

Результат

```
Value for myArray[0]:  3
Value for myArray[1]:  6
Value for myArray[2]:  9
Value for myArray[3]:  12
Value for myArray[4]:  15
0: 3
1: 6
2: 9
3: 12
4: 15
```

Анализ

Код листинга 13.1 создает массив, позволяет пользователю ввести значения для заполнения его элементов, а затем отображает их на экране. В строке 5 объявлен массив `myArray` из пяти целочисленных переменных. Как можно заметить, он объявлен с цифрой 5 в квадратных скобках. Это означает, что массив `myArray` может содержать

пять целочисленных значений. Каждый из элементов этого массива можно рассматривать как обычную целочисленную переменную.

В строке 7 начинается цикл от нуля до четырех, перебирающий все пять элементов массива. В строке 9 пользователю предлагается ввести значение, которое сохраняется в соответствующем элементе массива (строка 10).

Рассмотрим код строки 10 подробнее. Как можно заметить, для обращения к каждому элементу используется имя массива, сопровождаемого значением индекса, указанным в квадратных скобках. Каждый из этих элементов может быть впоследствии обработан подобно переменной типа массива.

Первое значение сохраняется в элементе массива `myArray[0]`, второе — в `myArray[1]`, и т.д. В строках 12 и 13 второй цикл `for` выводит каждое значение на экран.

> Обратите внимание: массивы начинаются с `0`, а не с `1`. Это обычная причина ошибок в программах, написанных новичками. Используя массивы, помните, что массив из десяти элементов начинается с элемента `ArrayName[0]` и заканчивается элементом `ArrayName[9]`, а элемент `ArrayName[10]` не используется.

Запись данных за пределами массива

При записи значения в элемент массива компилятор вычисляет необходимую область памяти на основании размера типа элемента и размера массива. Предположим, необходимо записать значение в переменную `LongArray[5]`, являющуюся шестым элементом массива. Компилятор умножает индекс (`5`) на размер переменной (в данном случае — `4`). Затем текущий указатель смещается на `20` байтов от начального адреса массива, и записывается новое значение.

Если попробовать записать значение в элемент `LongArray[50]`, то компилятор, проигнорировав тот факт, что такого элемента не существует, вычислит смещение от начала массива (200 байтов), а затем запишет значение по этому адресу. Здесь могут оказаться другие данные, и запись нового значения может иметь непредсказуемые последствия. Если повезет, программа зависнет сразу. Если нет, результаты проявятся намного позже и в совершенно другом месте. Обнаружить такую ошибку крайне сложно, даже если возникли подозрения, что что-то пошло не так.

Подобно слепому, которого попросили отнести предмет в дом номер шесть, компилятор двинется вдоль стенки от дома к дому, решив: “Необходимо отсчитать от первого дома (`MainStreet[0]`) пять зданий. Каждый дом — это четыре больших шага. Всего следует сделать 20 шагов”. Но если послать его на `MainStreet[100]`, размер которой всего 25 зданий, то, сделав 400 шагов, несчастный может угодить под грузовик. Так что будьте внимательны, отправляя его куда-либо.

Листинг 13.2 демонстрирует, что случается при записи данных за пределами массива.

> Не запускайте эту программу, система может зависнуть!

Листинг 13.2. Запись данных за пределами массива

```
0:  // Листинг 13.2. Демонстрация того, что случается при
1:  // записи данных за пределами массива
2:  #include <iostream>
3:  using namespace std;
4:
5:  int main()
6:  {
```

```
 7:     // стража
 8:     long sentinelOne[3];
 9:     long TargetArray[25]; // массив для заполнения
10:     long sentinelTwo[3];
11:     int i;
12:     for (i=0; i<3; i++)
13:     {
14:       sentinelOne[i] = 0;
15:       sentinelTwo[i] = 0
16:     }
17:     for (i=0; i<25; i++)
18:       TargetArray[i] = 10;
19:     // проверить текущее значение (должно быть 0)
20:     cout << "Test 1: \n";
21:     cout << "TargetArray[0]: "  << TargetArray[0]  << "\n";
22:     cout << "TargetArray[24]: " << TargetArray[24] << "\n\n";
23:
24:     for (i=0; i<3; i++)
25:     {
26:       cout << "sentinelOne["  << i << "]: ";
27:       cout <<  sentinelOne[i] << "\n";
28:       cout << "sentinelTwo["  << i << "]: ";
29:       cout <<  sentinelTwo[i] << "\n";
30:     }
31:
32:     cout << "\nAssigning...";
33:     for (i=0; i<=25; i++)    // Пройти чуть дальше, чем можно!
34:       TargetArray[i] = 20;
35:
36:     cout << "\nTest 2: \n";
37:     cout << "TargetArray[0]: "  << TargetArray[0]  << "\n";
38:     cout << "TargetArray[24]: " << TargetArray[24] << "\n";
39:     cout << "TargetArray[25]: " << TargetArray[25] << "\n\n";
40:     for (i=0; i<3; i++)
41:     {
42:       cout << "sentinelOne[" << i << "]: ";
43:       cout << sentinelOne[i]<< endl;
44:       cout << "sentinelTwo[" << i << "]: ";
45:       cout << sentinelTwo[i]<< endl;
46:     }
47:
48:     return 0;
49:   }
```

Результат

```
Test 1:
TargetArray[0]: 10
TargetArray[24]: 10

SentinelOne[0]: 0
SentinelTwo[0]: 0
SentinelOne[1]: 0
SentinelTwo[1]: 0
SentinelOne[2]: 0
SentinelTwo[2]: 0
```

```
Assigning...
Test 2:
TargetArray[0]: 20
TargetArray[24]: 20
TargetArray[25]: 20

SentinelOne[0]: 20
SentinelTwo[0]: 0
SentinelOne[1]: 0
SentinelTwo[1]: 0
SentinelOne[2]: 0
SentinelTwo[2]: 0
```

Анализ

В строках 8 и 10 объявляются два массива по три целых числа, которые выступают в качестве "стражи" массива `TargetArray`. Эти охранные массивы инициализированы нулевым значением в строках 12–16. В связи с тем, что один из массивов объявлен до охраняемого, а второй — после, в памяти они будут расположены в том же порядке: "стража" — по бокам, а основной массив — между ними. Поэтому при выходе массива `TargetArray` за пределы отведенного ему пространства будут изменены значения охранных массивов. Одни компиляторы рассчитывают память от первого адреса массива, другие — от последнего, поэтому и "охрану" приходится размещать с двух сторон.

Строки 20–30 подтверждают, что значения "стражей" при первой проверке равны нулю. В строке 34 всем элементам `TargetArray` присваивается значение `20`, но счетчик задан так, что значение присваивается и элементу `TargetArray[25]`, которого не существует.

Строки 37–39 выводят на экран значения `TargetArray` при второй проверке. Обратите внимание, что `TargetArray[25]` выдает на экран значение `20`. Но когда на экран выводятся значения массивов `SentinelOne` и `SentinelTwo`, то оказывается, что значение элемента `SentinelOne[0]` изменилось. Дело в том, что участок памяти, являющийся 25-ым элементом от начала массива `TargetArray`, — это тот же участок, в котором размещается элемент `SentinelOne[0]`. Если обратиться к несуществующему элементу `TargetArray[25]`, то на самом деле произойдет обращение к элементу `SentinelOne[0]`.

ПРИМЕЧАНИЕ

> Стоит отметить, что в связи с различиями в использовании памяти разными компиляторами результаты выполнения этой программы могут оказаться иными. Охранные массивы могут и не быть перезаписаны. Если дело обстоит так, попробуйте изменить строку 33 так, чтобы значения присваивались не 25-у, а 26-у элементу. Это увеличит вероятность перезаписи "стража". Безусловно, в результате может быть перезаписано что-нибудь еще, что приведет к нарушению работы системы.

Обнаружить эту ошибку иногда бывает очень трудно, поскольку значение `SentinelOne[0]` было изменено в той части кода, которая к массиву `SentinelOne` вообще никакого отношения не имеет.

Ошибка последнего столба

Ошибка записи данных за пределами массива встречается так часто, что для нее придумали термин — *ошибка последнего столба* (fence post error). Сколько столбов необходимо для десятиметровой изгороди, если ставить их через каждый метр? Большинство людей, не задумываясь, ответят: "Десять", но на самом деле необходимо одиннадцать столбов. Рис. 13.2 демонстрирует это наглядно.

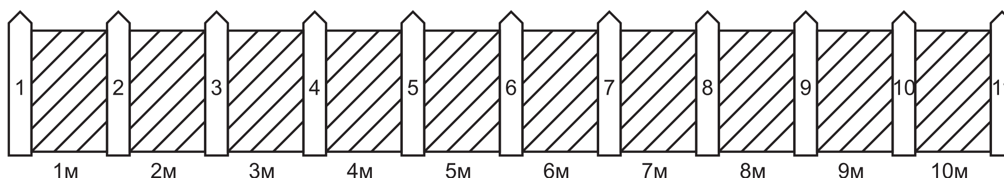

Рис. 13.2. Ошибка последнего столба

Подобный тип подсчета (“минус один”) отравляет жизнь любому начинающему программисту. Но со временем к этому можно привыкнуть и запомнить, что массив из 25-ти элементов заканчивается двадцать четвертым номером, а начинается нулевым.

ПРИМЕЧАНИЕ

> Некоторые программисты называют элемент `ArrayName[0]` нулевым. Это плохая привычка. Если элемент `ArrayName[0]` нулевой, то каким является `ArrayName[1]`? Первым? Если так, то, увидев надпись `ArrayName[24]`, сложно понять, что это не 24-ый элемент, а 25-ый. Поэтому правильнее говорить, что `ArrayName[0]` является первым элементом с нулевым номером.

Инициализация массивов

Небольшой массив переменных встроенных типов (например, `int` или `char`) можно инициализировать при объявлении. Для этого после имени массива помещают знак равенства (=) и заключенный в фигурные скобки список значений, отделяемых запятой. Например:

```
int IntegerArray[5] = { 10, 20, 30, 40, 50 };
```

Здесь объявлен массив `IntegerArray` из пяти целочисленных элементов, которым присвоены значения `IntegerArray[0]` — `10`, `IntegerArray[1]` — `20` и т.д.

Если размер массива не указан, но список значений присутствует, то будет создан и инициализирован массив достаточного размера, чтобы содержать все перечисленные значения. Таким образом, эта строка аналогична предыдущей:

```
int IntegerArray[] = { 10, 20, 30, 40, 50 };
```

Нельзя инициализировать количество элементов, превосходящее объявленный размер массива:

```
int IntegerArray[5] = { 10, 20, 30, 40, 50, 60 };
```

Такая строка приведет к ошибке во время компиляции, поскольку объявлен массив для пяти элементов, а инициализировать пытались шесть. Но следующая запись вполне допустима:

```
int IntegerArray[5] = {10, 20};
```

В данном случае объявлен массив из пяти элементов, а инициализированы только первые два: `IntegerArray[0]` и `IntegerArray[1]`.

Рекомендуется	Не рекомендуется
Позволить компилятору самостоятельно устанавливать размер инициализируемых массивов. **Называть** массивы осмысленными именами, как и все остальные переменные.	**Забывать**, что первый элемент массива имеет индекс, равный `0`. **Заносить** в массив больше элементов, чем объявлено.

Объявление массивов

В предыдущем коде для размеров массивов использовались “магические числа”: 3 — для охранных массивов и 25 — для массива TargetArray. Применение размеров констант обеспечивает большую надежность кода, поскольку при необходимости все значения можно изменить в одном месте.

Массивы могут иметь любые имена, допустимые для переменных. Имена массивов и переменных не могут совпадать в пределах одной области видимости. Следовательно, нельзя одновременно объявить массив по имени myCats[5] и переменную myCats.

Кроме того, при объявлении количества элементов вместо литералов можно использовать константу или перечисление. Фактически даже желательно использовать именно их, а не литеральное число, поскольку такой подход облегчает контроль над количеством элементов всех массивов, позволяя указать их размеры в едином месте. В листинге 13.2 использовались литеральные числа, но если размер массива TargetArray потребуется уменьшить до 20, то изменять придется несколько строк кода. Если использовать для этого константу, то достаточно изменить значение только одной константы.

Объявление количества элементов (или размера) массива при помощи перечисления осуществляется немного иначе. Это проиллюстрировано в листинге 13.3 на примере массива, который содержит значения дней недели.

Листинг 13.3. Использование перечислений и констант в массивах

```
 0:  // Листинг 13.3. Задание размера массива
 1:  // с помощью констант и перечислений
 2:
 3:  #include <iostream>
 4:  int main()
 5:  {
 6:    enum WeekDays { Sun, Mon, Tue, Wed,
 7:                    Thu, Fri, Sat, DaysInWeek };
 8:    int ArrayWeek[DaysInWeek] = { 10, 20, 30, 40, 50, 60, 70 };
 9:
10:    std::cout << "The value at Tuesday is: " << ArrayWeek[Tue];
11:    return 0;
12:  }
```

Результат

```
The value at Tuesday is: 30
```

Анализ

В строке 6 создано перечисление WeekDays (дни недели). Оно состоит из восьми членов: Sun (воскресенье) равно 0, а DaysInWeek (количество дней в неделе) — 7. В строке 8 объявлен массив по имени ArrayWeek, размер которого задан элементом DaysInWeek (со значением 7).

В строке 10 константа перечисления Tue (вторник) использована в качестве индекса элемента массива. Поскольку значением Tue является 2, на экран будет выведено значение третьего элемента массива (ArrayWeek[2]), равное 30.

Массивы

При объявлении массива сначала указывают тип хранимого объекта, а затем имя и количество элементов массива.

Пример 1:

```
int MyIntegerArray[90];
```

Пример 2:

```
long * ArrayOfPointersToLongs[100];
```

Для доступа к элементам массива используют оператор индекса.

Пример 1:

```
// Присвоение значения девятого элемента массива
// MyIntegerArray переменной theNinethInteger
int theNinethInteger = MyIntegerArray[8];
```

Пример 2:

```
// Присвоение значения девятого элемента массива
// ArrayOfPointersToLongs указателю pLong
long * pLong = ArrayOfPointersToLongs[8];
```

Массивы начинаются с нуля. Массив из *n* элементов пронумерован от 0 до *n-1*.

Массивы объектов

Массив может хранить любые объекты — как встроенные, так и созданные пользователем. При объявлении массива компилятору сообщают тип и количество хранимых объектов, что позволит выделить участок памяти требуемого размера. Компилятор определяет размер памяти, необходимой для одного элемента массива (объекта), на основании объявления класса. Чтобы объекты могли быть созданы при определении массива, класс должен обладать стандартным конструктором, которому не передают никаких аргументов.

Доступ к данным-членам объектов, находящихся в массиве, осуществляется в два этапа. Сначала, используя оператор индекса (`[]`), идентифицируют элемент массива, а затем с помощью точечного оператора (`.`) получают доступ к определенной переменной-члену. Листинг 13.4 демонстрирует создание массива из пяти объектов класса `Cat`.

Листинг 13.4. Создание массива объектов

```
 0:  // Листинг 13.4. Создание массива объектов
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  class Cat
 6:  {
 7:    public:
 8:      Cat() { itsAge = 1; itsWeight=5; }
 9:      ~Cat() {}
10:      int GetAge()    const { return itsAge; }
11:      int GetWeight() const { return itsWeight; }
12:      void SetAge(int age)  { itsAge = age; }
13:
```

```
14:     private:
15:       int itsAge;
16:       int itsWeight;
17:   };
18:
19:   int main()
20:   {
21:     Cat Litter[5];
22:     int i;
23:     for (i=0; i<5; i++)
24:       Litter[i].SetAge(2*i +1);
25:
26:     for (i=0; i<5; i++)
27:     {
28:       cout << "Cat #" << i+1<< ": ";
29:       cout << Litter[i].GetAge() << endl;
30:     }
31:     return 0;
32:   }
```

Результат

```
cat #1: 1
cat #2: 3
cat #3: 5
cat #4: 7
cat #5: 9
```

Анализ

Класс `Cat` объявлен в строках 5–17. Для создания массива объектов класс `Cat` должен иметь стандартный конструктор. В данном случае стандартный конструктор объявлен и определен в строке 8. Для каждого объекта класса `Cat` по умолчанию задается возраст `1` и вес `5`. Не забывайте, что при наличии любого собственного конструктора стандартный конструктор компилятором не создается. Поэтому необходимо создать собственный стандартный конструктор.

Первый цикл `for` (строки 23 и 24) устанавливает возраст каждого из пяти объектов класса `Cat` в массиве. Второй цикл `for` (строки 26–30) обращается к каждому из элементов и вызывает функцию `GetAge()`.

Чтобы вызвать метод `GetAge()` каждого из объектов класса `Cat`, приходится при обращении к элементу массива `Litter[i]` использовать точечный оператор (`.`) и имя функции-члена.

Многомерные массивы

Можно создать массив более одной размерности. Каждая размерность представляет собой дополнительный индекс массива. Следовательно, двумерный массив имеет два индекса, трехмерный — три и т.д. Массив может иметь любое количество размерностей, но в большинстве случаев достаточно одной или двух.

Примером двумерного массива может служить шахматная доска. Одна размерность представляет собой восемь рядов по горизонтали, другая — восемь рядов по вертикали (рис. 13.3).

Предположим, существует класс `SQUARE` (клетка). Объявление массива `Board` (доска) выглядело бы следующим образом:

```
SQUARE Board[8][8];
```

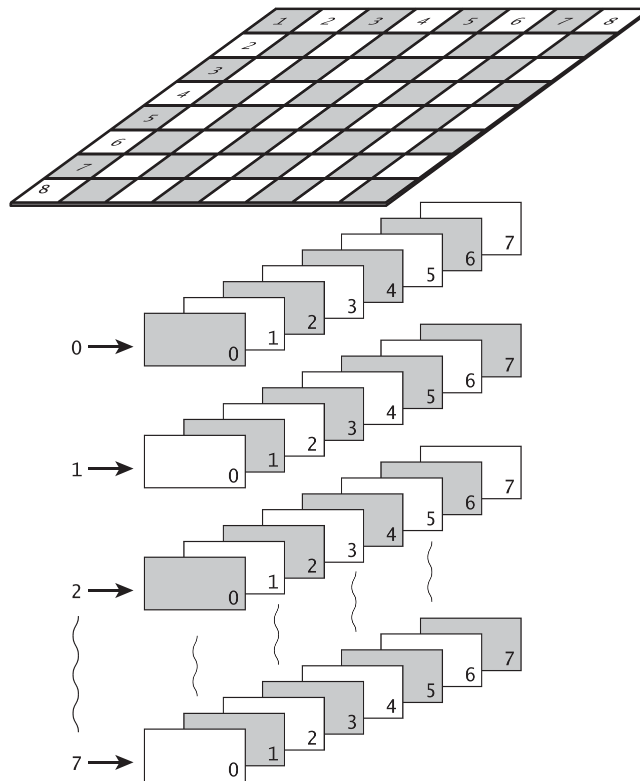

Рис. 13.3. Шахматная доска как двумерный массив

Те же данные можно представить в виде одномерного массива на 64 клетки, например:

```
SQUARE Board[64];
```

Но это не будет соответствовать реальному объекту. В начале игры король стоит на четвертой клетке в первом ряду, эта позиция соответствует первому элементу по одному индексу и четвертому — по второму.

```
Board[0][3];
```

Инициализация многомерных массивов

Многомерные массивы также можно инициализировать. Элементам массива присваивается список значений таким образом, что вслед за последним значением первого индекса массива начинается первое значение второго индекса, и т.д.[1]

```
int theArray[5][3];
```

Следовательно, если объявлен такой массив, то первые три элемента входят в элемент `theArray[0]`, следующие три — в элемент `theArray[1]`, и т.д.

А инициализируется этот массив следующим образом:

```
int theArray[5][3] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 };
```

---

[1] *Т.е. двумерный массив разворачивается построчно.* — Прим. ред.

Чтобы было понятнее, значения при инициализации можно разделить фигурными скобками, например:

```
int theArray[5][3] = {
  { 1, 2, 3},
  { 4, 5, 6},
  { 7, 8, 9},
  {10,11,12},
  {13,14,15} };
```

Компилятор проигнорирует внутренние фигурные скобки, но они сделают набор чисел нагляднее и понятнее.

Вне зависимости от фигурных скобок каждое значение следует отделять запятой. Весь набор значений для инициализации расположен во внешних фигурных скобках и заканчивая точкой с запятой.

В листинге 13.5 создается двумерный массив. Первая размерность — набор чисел от 0 до 4, а вторая размерность состоит из удвоенного значения в первой.

Листинг 13.5. Создание многомерного массива

```
 0:  // Листинг 13.5. Создание многомерного массива
 1:  #include <iostream>
 2:  using namespace std;
 3:
 4:  int main()
 5:  {
 6:    int SomeArray[2][5] = { {0,1,2,3,4}, {0,2,4,6,8} },
 7:    for (int i=0; i<2; i++)
 8:    {
 9:      for (int j=0; j<5; j++)
10:      {
11:        cout << "SomeArray[" << i << "][" << j << "]: ";
12:        cout << SomeArray[i][j]<< endl;
13:      }
14:    }
15:    return 0;
16:  }
```

Результат

```
SomeArray[0][0]: 0
SomeArray[0][1]: 1
SomeArray[0][2]: 2
SomeArray[0][3]: 3
SomeArray[0][4]: 4
SomeArray[1][0]: 0
SomeArray[1][1]: 2
SomeArray[1][2]: 4
SomeArray[1][3]: 6
SomeArray[1][4]: 8
```

Анализ

В строке 6 объявлен двумерный массив SomeArray. Первый ряд элементов состоит из двух целых чисел, а второй — из пяти. Это создает сетку из 2×5 элементов (рис. 13.4).

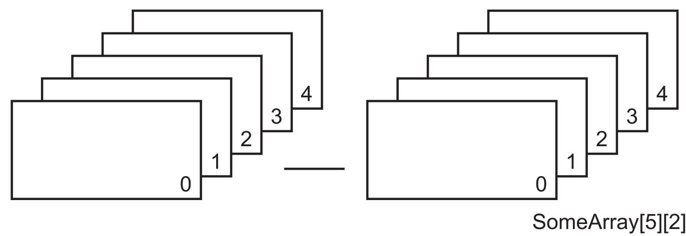

Рис. 13.4. Массив 2×5

Значения разделены на два набора чисел. Первый набор — это исходные числа, а второй — их удвоенные значения. В этом коде исходные значения заданы непосредственно, но их также можно было вычислить. Строки 7 и 9 содержат два цикла `for`. Внешний цикл `for` (начинающийся в строке 7) перебирает все элементы по первой размерности (т.е. оба набора целых чисел). Для каждого элемента этой размерности (набора) внутренний цикл `for` (начинающийся в строке 9) перебирает все элементы второй размерности. Это сопровождается выводом значений на экран. За элементом `SomeArray[0][0]` следует элемент `SomeArray[0][1]` и т.д. Приращение значения по первой размерности происходит только после полного перебора второй. Затем перебор второй размерности начинается снова.

Немного о памяти

При объявлении массива компилятору точно указывают, сколько объектов планируется в нем сохранить. Компилятор зарезервирует память для всех объектов массива, даже если они не будут использованы. Если известно заранее, сколько элементов должен хранить массив, то никаких проблем не возникнет. Например, шахматная доска всегда имеет 64 клетки, а у кошки не может быть больше 10 котят. Но если количество элементов массива неизвестно, придется применить иные средства организации данных.

В этой книге рассматриваются массивы указателей, массивы, находящиеся в динамической памяти, и другие типы структур. Но более подробная информация по этой теме приведена в предыдущей книге автора, *C++ Unleashed*, опубликованной издательством *Sams Publishing*, а также в приложении Д, “Связанные списки”.

Массивы указателей

До сих пор обсуждались массивы, члены которых размещались в стеке. Но размер стека значительно меньше объема динамической памяти, поэтому можно объявить объекты массива в динамической памяти, а в самом массиве хранить лишь указатели на них. Этот существенно уменьшает объем используемой стековой памяти. Листинг 13.6 является модификацией листинга 13.4, но элементы массива в нем расположены в динамической памяти. Поскольку появилась возможность занять больше памяти, размер массива увеличен с 5 до 500 элементов, а имя массива `Litter` (потомство) изменено на `Family` (семья).

Листинг 13.6. Размещение массива в динамической памяти

```
0:  // Листинг 13.6. Массив указателей на объекты
1:
2:  #include <iostream>
3:  using namespace std;
```

```
 4:
 5:  class Cat
 6:  {
 7:    public:
 8:      Cat() { itsAge = 1; itsWeight=5; }
 9:      ~Cat() {}                                 // деструктор
10:      int  GetAge()    const { return itsAge; }
11:      int  GetWeight() const { return itsWeight; }
12:      void SetAge(int age)   { itsAge = age; }
13:
14:    private:
15:      int itsAge;
16:      int itsWeight;
17:  };
18:
19:  int main()
20:  {
21:    Cat * Family[500];
22:    int i;
23:    Cat * pCat;
24:    for (i=0; i<500; i++)
25:    {
26:      pCat = new Cat;
27:      pCat->SetAge(2*i+1);
28:      Family[i] = pCat;
29:    }
30:
31:    for (i=0; i<500; i++)
32:    {
33:      cout << "Cat #" << i+1 << ": ";
34:      cout << Family[i]->GetAge() << endl;
35:    }
36:    return 0;
37:  }
```

Результат

```
Cat #1: 1
Cat #2: 3
Cat #3: 5
...
Cat #499: 997
Cat #500: 999
```

Анализ

Класс Cat, объявленный в строках 5–17, идентичен классу Cat, объявленному в листинге 13.4. А массив Family, объявленный в строке 21, на сей раз будет содержать до 500 элементов, являющихся *указателями* на объекты класса Cat.

Цикл инициализации (строки 24–29) создает в динамической памяти 500 новых объектов класса Cat, и для каждого из них устанавливается возраст вдвое больше его номера, увеличенного на единицу. Следовательно, первый кот будет иметь возраст 1, второй — 3, третий — 5 и т.д. Затем указатели добавляются в массив.

Второй цикл (строки 31–35) выводит каждое значение на экран. Код строки 33 отображает на экране число, соответствующее номеру объекта. Поскольку индексирование начинается с нуля, для правильного счета в строке 33 к числу добавляется 1.

Для доступа к указателю используется индекс массива Family[i], а полученный адрес используется для доступа к методу GetAge() текущего объекта.

В данном случае сам массив Family и все его указатели хранятся в стеке, но 500 объектов класса Cat созданы и размещены в области динамической памяти.

Арифметические операции над указателями

На занятии дня 8, "Указатели", первоначальные сведения об указателях уже были представлены. Однако прежде, чем продолжать изучение массивов, имеет смысл вернуться к указателям и рассмотреть еще одну тему — арифметические операции над указателями.

С указателями можно осуществить несколько математических действий. Их можно вычитать друг из друга. Давайте рассмотрим одну из возможностей применения этого подхода. Если существуют два указателя, содержащих адреса двух разных элементов массива, то их разница позволит выяснить, сколько элементов расположено между ними. При анализе символьных массивов это может оказаться весьма полезным, как проиллюстрировано в листинге 13.7.

Листинг 13.7. Анализ и разбиение символьной строки на слова

```
 0:  #include <iostream>
 1:  #include <ctype.h>
 2:  #include <string.h>
 3:
 4:  bool GetWord(char* theString,
 5:               char* word, int& wordOffset);
 6:
 7:  // Основная программа
 8:  int main()
 9:  {
10:    const int bufferSize = 255;
11:    char buffer[bufferSize+1];    // содержит всю строку
12:    char word[bufferSize+1];      // содержит слово
13:    int wordOffset = 0;           // начальная позиция
14:
15:    std::cout << "Enter a string: ";
16:    std::cin.getline(buffer,bufferSize);
17:
18:    while (GetWord(buffer, word, wordOffset))
19:    {
20:      std::cout << "Got this word: " << word << std::endl;
21:    }
22:    return 0;
23:  }
24:
25:  // Функция разделения строки на слова.
26:  bool GetWord(char* theString, char* word, int& wordOffset)
27:  {
28:    if (theString[wordOffset] == 0)  // Конец строки?
29:      return false;
30:
31:    char *p1, *p2;
32:    p1 = p2 = theString+wordOffset;  // указатель на следующее
33:                                     // слово
```

```
34:     // Убрать предваряющие пробелы
35:     for (int i=0; i<(int)strlen(p1) && !isalnum(p1[0]); i++)
36:       p1++;
37:
38:     // Получено ли слово?
39:     if (!isalnum(p1[0]))
40:       return false;
41:
42:     // Теперь p1 указывает на начало следующего слова.
43:     // p2 - теперь тоже.
44:     p2 = p1;
45:
46:     // Перевести p2 в конец слова
47:     while (isalnum(p2[0]))
48:       p2++;
49:
50:     // Теперь p2 - в конце слова,
51:     // p1 - в начале слова,
52:     // а разница между ними - это длина слова.
53:     int len = int (p2 - p1);
54:
55:     // Скопировать слово в буфер
56:     strncpy (word, p1, len);
57:
58:     // Добавить пустой завершающий символ.
59:     word[len] = '\0';
60:
61:     // Теперь найти начало следующего слова.
62:     for (int j=int(p2-theString); j<(int)strlen(theString)
63:          && !isalnum(p2[0]); j++)
64:     {
65:       p2++;
66:     }
67:
68:     wordOffset = int(p2-theString);
69:
70:     return true;
71:   }
```

Результат

```
Enter a string: this code first appeared in C++ Report
Got this word: this
Got this word: code
Got this word: first
Got this word: appeared
Got this word: in
Got this word: C
Got this word: Report
```

Анализ

Эта программа предлагает пользователю ввести предложение, которое она разделяет на отдельные слова (наборы алфавитно-цифровых символов). Предложение ввести строку расположено в строке 15. В строке 18 эта строка передается функции `GetWord()` наряду с предназначенным для хранения первого слова буфером и целочисленной переменной `WordOffset`, инициализированной в строке 13 нулевым значением.

Функция GetWord() возвращает отдельные слова из строки до тех пор, пока не будет достигнут конец строки. Пока функция GetWord() не вернет значение False, возвращаемые ей слова отображаются на экране кодом строки 20.

После каждого вызова функции GetWord() управление возвращается к строке 26. В строке 28 осуществляется проверка значения string[wordOffset] на равенство нулю. Это произойдет в случае, если процесс находится в конце строки (функция GetWord() возвратила значение False). Функция cin.GetLine() позволяет завершить введенную строку пустым символом, т.е. символом, полученным в результате вычисления управляющей последовательности '\0';.

В строке 31 объявлены два указателя на тип char (символ) p1 и p2, а в строке 32 им присваиваются адрес начала строки плюс смещение wordOffset. Поскольку первоначально значением переменной wordOffset является нуль, они указывают на начало строки.

Строки 35 и 36 содержат цикл, перебирающий строку в поисках первого алфавитно-цифрового символа, увеличивая соответственно значение указателя p1. Код строк 39 и 40 проверяет, является ли найденный символ алфавитно-цифровым. Если это не так, возвращается значение False.

Теперь указатель p1 содержит адрес начала следующего слова, а в строке 44 указатель p2 устанавливается в ту же позицию.

Цикл в строках 47 и 48, перебирая слово, увеличивает значение указателя p2 и останавливается на первом, не алфавитно-цифровом, символе. Теперь указатель p2 содержит адрес конца того слова, на начало которого указывает p1. Вычислив разницу значений в указателях p1 из p2 и приведя результат в строке 53 к целому числу, получим длину слова. Передав методу strncpy() из стандартной библиотеки буфер word, отправную точку p1 и длину, скопируем слово в буфер.

В строке 59 к расположенному в буфере слову добавляется нулевой символ, отмечающий конец слова. Затем значение адреса в указателе увеличивается на единицу, чтобы он указывал на начало следующего слова, а значение его смещения заносится в переменную wordOffset. И, наконец, возвращается значение true, свидетельствующее об успешном обнаружении слова.

Это классический пример кода, изучать который удобнее всего в режиме пошагового выполнения в отладчике.

В этом листинге можно заметить несколько случаев арифметических операций над указателями: в строке 53 вычитание одного указателя из другого позволяет вычислить количество элементов между двумя указателями, а в строке 55 приращение значения указателя применяется для его сдвига на следующий элемент массива. Использование арифметических операций над указателями — это общепринятый подход при работе с указателями и массивами. Однако поскольку ошибки здесь способны привести к опасным последствиям, применять его следует внимательно.

Объявление массивов в области динамической памяти

Весь массив можно разместить в области динамической памяти (известной также под именем heap). Для этого при объявлении массива используется оператор new. Результатом окажется указатель на пространство в динамической памяти, где и будет содержаться массив, например:

```
Cat *Family = new Cat[500];
```

Здесь объявлено, что Family будет указателем на первый элемент массива из 500 объектов класса Cat. Иными словами, Family указывает на элемент Family[0] (или содержит его адрес).

Преимуществом такого способа использования массива является возможность арифметических операций над указателями для доступа к каждому элементу массива `Family`. Например, можно написать:

```
Cat *Family = new Cat[500];
Cat *pCat = Family;          // pCat указывает на Family[0]
pCat->SetAge(10);            // присвоить Family[0] значение 10
pCat++;                      // вперед к Family[1]
pCat->SetAge(20);            // присвоить Family[1] значение 20
```

Здесь в динамической памяти объявлен массив из 500 объектов класса `Cat`, а также создан указатель, содержащий адрес начала массива. С помощью этого указателя осуществляется вызов метода `SetAge()` первого из объектов класса `Cat`, который и присваивает ему значение `10`. Затем указатель увеличивается и указывает уже на следующий объект класса `Cat`, поэтому при вызове метода `SetAge()` значение `20` будет присвоено второму объекту.

Указатель на массив и массив указателей

Рассмотрим следующие три объявления:

```
1:  Cat FamilyOne[500];
2:  Cat * FamilyTwo[500];
3:  Cat * FamilyThree = new Cat[500];
```

Здесь `FamilyOne` — это массив из 500 объектов класса `Cat`, `FamilyTwo` — массив из 500 указателей на объекты класса `Cat`, а `FamilyThree` — указатель на массив из 500 объектов класса `Cat`.

Различие между этими тремя строками весьма существенно и влияет на способ существования массивов. Но самое удивительное то, что указатель `FamilyThree` является вариантом `FamilyOne`, а от указателя `FamilyTwo` отличается принципиально.

В этом вся суть проблемы взаимосвязи указателей и массивов. В третьем случае `FamilyThree` представляет собой указатель на массив. Т.е. адрес, находящийся в указателе `FamilyThree`, является адресом первого элемента в этом массиве. Но это аналогично тому, что имеет место и для `FamilyOne`!

Имена массивов и указателей

В языке C++ имя массива является постоянным указателем на первый элемент массива.

```
Cat Family[50];
```

Следовательно, здесь `Family` представляет собой указатель на переменную `&Family[0]`, являющуюся первым элементом массива `Family`.

Вполне допустимо использовать имена массивов как постоянные указатели и наоборот. Следовательно, `Family + 4` — вполне законный способ доступа к данным в элементе `Family[4]`.

При инкременте и декременте (увеличении и уменьшении) указателя компилятор делает все вычисления сам. Адрес, полученный в результате вычисления выражения `Family + 4`, не на четыре байта больше исходного, а на четыре объекта. Если все объекты обладают размером четыре байта, то `Family + 4` составит 16 байтов от начала массива. Таким образом, если каждый объект класса `Cat` будет содержать четыре переменные-члена типа `long` размером четыре байта каждая и две — типа `short` размером два байта каждая, то размер каждого объекта класса `Cat` составит 20 байтов, а `Family + 4` будет на 80 байтов больше `Family` (адреса начала массива).

Листинг 13.8 иллюстрирует объявление и использование массива в динамической памяти.

Листинг 13.8. Создание массива в динамической памяти

```
 0:  // Листинг 13.8. Массив в динамической памяти
 1:
 2:  #include <iostream>
 3:
 4:  class Cat
 5:  {
 6:    public:
 7:      Cat() { itsAge=1; itsWeight=5; }
 8:      ~Cat();
 9:      int  GetAge()    const { return itsAge; }
10:      int  GetWeight() const { return itsWeight; }
11:      void SetAge(int age)   { itsAge = age; }
12:
13:    private:
14:      int itsAge;
15:      int itsWeight;
16:  };
17:
18:  Cat :: ~Cat()
19:  {
20:    // std::cout << "Destructor called!\n";
21:  }
22:
23:  int main()
24:  {
25:    Cat * Family = new Cat[500];
26:    int i;
27:
28:    for (i=0; i<500; i++)
29:    {
30:      Family[i].SetAge(2*i +1);
31:    }
32:
33:    for (i=0; i<500; i++)
34:    {
35:      std::cout << "Cat #" << i+1 << ": ";
36:      std::cout << Family[i].GetAge() << std::endl;
37:    }
38:
39:    delete [] Family;
40:
41:    return 0;
42:  }
```

Результат

```
Cat #1: 1
Cat #2: 3
Cat #3: 5
...
Cat #499: 997
Cat #500: 999
```

Анализ

В строке 25 объявлен массив `Family`, содержащий 500 объектов класса `Cat`. Весь массив создан в динамической памяти с помощью выражения `new Cat[500]`.

Как можно заметить, в строке 30 объявленный указатель применяется с использованием оператора индекса `[]`, а, следовательно, обрабатывается так же, как и обычный массив. В строке 36 при вызове метода `GetAge()` это продемонстрировано еще раз. Таким образом, при решении практических задач указатель на массив `Family` можно использовать вместо имени массива. Не забывайте, однако, освободить память, выделенную для массива при его создании, как это сделано в строке 39 при помощи оператора `delete`.

Удаление массивов из динамической памяти

Что происходит с выделенной для объектов класса `Cat` памятью, когда массив удаляется? Не происходит ли утечка памяти?

Удаление массива `Family` автоматически возвращает всю выделенную для него память, если оператор `delete` использован с квадратными скобками `[]`. Компилятор вычисляет размер всех элементов массива и освобождает занимаемую им область динамической памяти.

Чтобы продемонстрировать это, уменьшим в строках 25, 28 и 33 размер массива с 500 до 10-ти элементов, а затем раскомментируем в строке 20 оператор `cout`. По достижении строки 39 массив будет удален, и для каждого объекта класса `Cat` будет вызван деструктор.

Когда с помощью оператора `new` в динамической памяти создается элемент, то при его удалении непременно следует освободить занимаемую им область памяти. Создав в динамической памяти массив с помощью оператора `new <класс>[размер]`, впоследствии необходимо применить оператор `delete[]`, чтобы освободить эту область памяти. Квадратные скобки сообщают компилятору о том, что удаляется массив.

Если забыть квадратные скобки, то освобожден будет лишь первый объект массива. В этом легко убедиться, удалив скобки в строке 39. Если отредактировать строку 20 так, чтобы при каждом вызове деструктора печаталось сообщение, то можно увидеть, что был освобожден только один объект класса `Cat`. Поздравляем! Только что произошла утечка памяти.

Изменение размера массива во время выполнения

Наибольшим преимуществом создания массива в распределяемой памяти является возможность выяснить необходимый для него размер во время выполнения, а затем создать его. Например, если запросить у пользователя размер семьи и занести его значение в переменную `SizeOfFamily` (размер семьи), то массив элементов класса `Cat` можно объявить следующим образом:

```
Cat *pFamily = new Cat[SizeOfFamily];
```

В результате будет получен указатель на массив объектов класса `Cat`. Теперь можно создавать указатель на первый элемент массива, а затем, используя этот указатель и арифметические операции над указателями, перебрать в цикле все его элементы следующим образом:

```
Cat *pCurrentCat = Family[0];
for (int Index=0; Index<SizeOfFamily; Index++, pCurrentCat++)
```

```
{
  pCurrentCat->SetAge(Index);
};
```

Поскольку язык C++ рассматривает массивы как частный случай указателя, второй указатель можно отбросить и просто использовать стандартную индексацию массива:

```
for (int Index=0; Index<SizeOfFamily; Index++)
{
  pFamily[Index].SetAge(Index);
};
```

Использование квадратных скобок позволяет компилятору автоматически получить адрес соответствующего указателя и произвести необходимые арифметические операции над ним.

Еще одно преимущество: подобный подход можно использовать для изменения размера массива во время выполнения, если отведенный для него участок памяти исчерпан. Такую возможность демонстрирует листинг 13.9.

Листинг 13.9. Изменение размера массива во время выполнения

```
 0:  //Listing 13.9. Изменение размера массива во время выполнения
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:  int main()
 5:  {
 6:    int  AllocationSize = 5;
 7:    int *pArrayOfNumbers = new int[AllocationSize];
 8:    int  ElementsUsedSoFar = 0;
 9:    int  MaximumElementsAllowed = AllocationSize;
10:    int  InputNumber = -1;
11:
12:    cout << endl << "Next number = ";
13:    cin >> InputNumber;
14:
15:    while (InputNumber>0)
16:    {
17:      pArrayOfNumbers[ElementsUsedSoFar++] = InputNumber;
18:
19:      if (ElementsUsedSoFar == MaximumElementsAllowed)
20:      {
21:        int *pLargerArray =
22:          new int[MaximumElementsAllowed+AllocationSize];
23:
24:        for (int CopyIndex=0;
25:             CopyIndex<MaximumElementsAllowed;
26:             CopyIndex++)
27 :       {
28:          pLargerArray[CopyIndex] = pArrayOfNumbers[CopyIndex];
29:        };
30:
31:        delete [] pArrayOfNumbers;
32:        pArrayOfNumbers = pLargerArray;
33:        MaximumElementsAllowed += AllocationSize;
34:      };
35:      cout << endl << "Next number = ";
36:      cin >> InputNumber;
37:    }
```

```
38:
39:    for (int Index=0; Index<ElementsUsedSoFar; Index++)
40:    {
41:      cout << pArrayOfNumbers[Index] << endl;
42:    }
43:    return 0;
44:  }
```

Результат

```
Next number = 10
Next number = 20
Next number = 30
Next number = 40
Next number = 50
Next number = 60
Next number = 70
Next number = 0
10
20
30
40
50
60
70
```

Анализ

В этом примере запрашиваются и сохраняются в массиве числа, введенные одно за другим. При вводе числа, равного или меньше 0, собранный массив чисел отображается на экране.

Рассмотрим этот код подробнее. В строках 6—9 объявлены несколько переменных. А именно: в строке 6 задан исходный размер массива (5), код строки 7 создает массив в распределяемой памяти, а его адрес присваивает указателю pArrayOfNumbers.

Код строк 12—13 запрашивает у пользователя первое число и помещает его в переменную InputNumber. Если введенное значение больше нуля, в строке 15 начинается его обработка; в противном случае управление переходит к строке 38.

В строке 17 значение переменной InputNumber помещается в массив. Сначала все происходит без проблем, поскольку выделенного участка памяти еще хватает. Код строки 19 проверяет, не является ли текущий элемент последним (т.е. обладает ли массив свободными ячейками). Если свободный участок памяти имеется, управление проходит к строке 35, в противном случае выполняется код тела оператора if, позволяющий увеличить размер массива (строки 20—34).

Новый массив создается в строке 21. Его размер будет на пять элементов (AllocationSize) больше, чем текущего. Затем код строк 24—29 копирует содержимое старого массива в новый. Здесь использована стандартная для массива форма записи, но можно было использовать также и арифметические операции над указателями.

Код строки 31 удаляет старый массив, а код строки 32 заменяет старый указатель указателем на б*о*льший массив. Строка 33 увеличивает значение переменной MaximumElementsAllowed так, чтобы оно соответствовало новому размеру.

Код строк 39—42 отображает полученный в результате массив на экране.

Рекомендуется	Не рекомендуется
Помнить, что массив из *n* элементов пронумерован от 0 до *n-1*. **Применять** для доступа к элементам массива индексы, а для доступа к указателям массива — точечный оператор. **Использовать** оператор `delete[]`, чтобы удалить весь массив, созданный в динамической памяти. Оператор `delete` без квадратных скобок удалит только первый элемент массива.	**Записывать** и читать данные вне пределов массива. **Путать** массив указателей с указателем на массив. **Забывать** освободить память, выделенную при помощи оператора `new`.

Массивы символов и строки

Строка в стиле C представляет собой массив символов, завершающийся пустым значением (`null`). До сих пор в этой книге единственными строками в стиле C были безымянные строковые константы, например:

```
cout << "hello world.\n";
```

Строку стиля C можно объявить и инициализировать, как и любой другой массив:

```
char Greeting[] =
{ 'H', 'e', 'l', 'l', 'o', ' ', 'W','o','r','l','d', '\0' };
```

В данном случае объявлен массив символов `Greeting`, который инициализирован набором символов. Последний символ, `'\0'`, является пустым (символом `null`). Именно он служит для функций языка C++ признаком конца строки. Хотя такой "посимвольный" подход и работоспособен, но труден для вывода и порождает слишком много ошибок. Язык C++ допускает использование более кратких форм. Например, объявление предыдущей строки может выглядеть так:

```
char Greeting[] = "Hello World";
```

Обратите внимание на следующие две особенности такого синтаксиса:

- вместо отдельных символов в одинарных кавычках, разделенных запятыми и окруженных фигурными скобками, применяются лишь двойные кавычки;
- добавлять символ `null` в конце строки не нужно, компилятор сделает это сам.

При объявлении строковой переменной необходимо удостовериться, что ее размер достаточен для выполнения поставленной задачи. Длину строки в стиле C составляет количество символов строки, включая символы пробела и завершающий нулевой символ. Например, строка "`Hello World`" в стиле C занимает 12 байтов: `Hello` — 5 байтов, пробел — 1, `World` — 5 и символ `null` — еще один.

Можно также создавать и неинициализированные символьные массивы. Однако при этом следует удостовериться, что в буфер будет записано данных не больше его вместимости.

Листинг 13.10 демонстрирует использование неинициализированного буфера.

Листинг 13.10. Заполнение массива

```
0:  // Листинг 13.10. Массив как символьный буфер
1:
2:  #include <iostream>
```

```
 3:
 4:  int main()
 5:  {
 6:    char buffer[80];
 7:    std::cout << "Enter the string: ";
 8:    std::cin >> buffer;
 9:    std::cout << "Here is's the buffer:  "
                << buffer << std::endl;
10:    return 0;
11:  }
```

Результат

```
Enter the string: Hello World
Here's the buffer:  Hello
```

Анализ

В строке 6 объявлен буфер размером 80 символов. Он достаточно велик, чтобы содержать 79-символьную строку в стиле С и завершающий символ `null`.

В строке 7 пользователю предлагают ввести строку, которая будет (в строке 8) введена в буфер. Концевой символ `null` добавляется оператором `cin`.

Здесь возникают две проблемы (см. листинг 13.10). Во-первых, если пользователь вводит строку длиннее 79-ти символов, то оператор `cin` запишет данные за пределами буфера. Во-вторых, если пользователь введет пробел, то `cin` воспримет его как конец строки и остановит запись в буфер.

Для разрешения этих проблем необходимо создать специальный метод `cin.get()`, которому передают три параметра:

- буфер для заполнения;
- максимальное количество символов;
- символ для завершения ввода.

По умолчанию критерием завершения ввода является символ новой строки. Листинг 13.11 демонстрирует применение этого метода.

Листинг 13.11. Заполнение массива максимальным количеством символов

```
 0:  // Листинг 13.11. Применение метода cin.get()
 1:
 2:  #include <iostream>
 3:  using namespace std;
 4:
 5:  int main()
 6:  {
 7:    char buffer[80];
 8:    cout << "Enter the string: ";
 9:    cin.get(buffer, 79);      // больше 79 или новая строка
10:    cout << "Here's the buffer:  " << buffer << endl;
11:    return 0;
12:  }
```

Результат

```
Enter the string: Hello World
Here's the buffer: Hello World
```

Анализ

В строке 9 расположен вызов метода `cin.get()`. Буфер, объявленный в строке 7, передается в качестве первого аргумента. Второй аргумент — максимальное количество вводимых символов. В данном случае — 79, чтобы учесть завершающий символ `null`. Третий параметр (символ завершения ввода) необязателен, поскольку по умолчанию признаком завершения является новая строка.

При вводе пробелов, символов табуляции или других непечатаемых символов они также войдут в состав строки. Символом новой строки заканчивается ввод. Ввод 79-ти символов также приведет к завершению ввода. Это можно проверить, повторно запустив код и попытавшись ввести строку, длиннее 79-ти символов.

Функции `strcpy()` и `strncpy()`

Язык C++ унаследовал от языка C библиотеку функций для строковых операций. Существует множество встроенных функций, две из них осуществляют копирование одной строки в другую. Это функции `strcpy()` и `strncpy()`. Функция `strcpy()` копирует содержимое строки в указанный буфер, а функция `strncpy()` копирует определенное количество символов из одной строки в другую. Листинг 13.12 демонстрирует применение функции `strcpy()`.

Листинг 13.12. Использование функции `strcpy()`

```
 0:  // Листинг 13.12. Использование функции strcpy()
 1:
 2:  #include <iostream>
 3:  #include <string.h>
 4:  using namespace std;
 5:
 6:  int main()
 7:  {
 8:    char String1[] = "No man is an island";
 9:    char String2[80];
10:
11:    strcpy(String2,String1);
12:
13:    cout << "String1: " << String1 << endl;
14:    cout << "String2: " << String2 << endl;
15:    return 0;
16:  }
```

Результат

```
String1: No man is an island
String2: No man is an island
```

Анализ

Файл заголовка `string.h` подключен в строке 3. Этот файл содержит прототип функции `strcpy()`, принимающей два символьных массива: результирующий и исходный. Если исходный массив окажется больше результирующего, функция `strcpy()` осуществит запись за пределами результирующего буфера.

Во избежание этой ошибки стандартная библиотека располагает функцией `strncpy()`. Этому варианту передают максимальное количество копируемых символов.

Функция strncpy() осуществляет копирование до первого символа null или максимального количества символов, определенного для результирующего буфера. Листинг 13.13 демонстрирует применение функции strncpy().

Листинг 13.13. Использование функции strncpy()

```
 0:  // Листинг 13.13. Использование функции strncpy()
 1:
 2:  #include <iostream>
 3:  #include <string.h>
 4:
 5:  int main()
 6:  {
 7:    const int MaxLength = 80;
 8:    char String1[] = "No man is an island";
 9:    char String2[MaxLength+1];
10:
11:    strncpy(String2,String1,MaxLength);
12:
13:    std::cout << "String1: " << String1 << std::endl;
14:    std::cout << "String2: " << String2 << std::endl;
15:    return 0;
16:  }
```

Результат

```
String1: No man is an island
String2: No man is an island
```

Анализ

Еще один простой пример кода. Подобно предыдущему листингу, здесь данные из одной строки просто копируются в другую. В строке 11 обращение к функции strcpy() было заменено на обращение к функции strncpy(), которой передают третий параметр: максимальное количество символов для копирования. Буфер String2 объявлен как массив из MaxLength+1 символов. Дополнительный элемент предназначен для символа null, который обе функции, strcpy() и strncpy(), добавляют в конец строки автоматически.

> Подобно продемонстрированным в листинге 13.9 целочисленным массивам, размеры символьных массивов также можно изменить, разместив в распределяемой памяти новый экземпляр массива и скопировав в него элементы прежнего.
>
> В наиболее гибких классах строк C++, позволяющих программистам увеличивать и уменьшать их размер, а также вставлять или удалять элементы из середины строк, используется похожий подход.

Строковые классы

Язык C++ унаследовал завершающий строку символ null от языка C вместе с содержащей функцию strcpy() библиотекой функций, но эти функции не интегрированы в объектно-ориентированную среду. Стандартная библиотека содержит класс String, инкапсулирующий специальный набор данных и функций для управления ими. Открыты лишь функции доступа, а сами данные класса String от клиента скрыты.

В качестве упражнения создадим собственный специальный класс `String`. Этот класс должен преодолеть исходные ограничения символьных массивов. Подобно всем остальным массивам, символьные массивы статичны. Их размер задается при объявлении, и независимо от того, какое количество элементов массива используется, размер занимаемого участка памяти остается неизменным, а запись за пределами массива чревата неприятностями.

> Создаваемый класс `String`, безусловно, будет иметь ограниченные возможности и не может применяться в коммерческих целях. Тем более, что стандартная библиотека располагает полной и надежной версией класса `String`.

Грамотно разработанный класс `String` занимает столько памяти, сколько необходимо для хранения переданных ему данных. Если класс не сможет выделить достаточного количества памяти, следует предусмотреть элегантный выход из этой ситуации.

Листинг 13.14 представляет реализацию класса `String` в первом приближении.

Листинг 13.14. Использование класса `String`

```
 0:  // Листинг 13.14. Использование класса String
 1:
 2:  #include <iostream>
 3:  #include <string.h>
 4:  using namespace std;
 5:
 6:  // Рудиментарный класс string
 7:  class String
 8:  {
 9:    public:
10:      // конструкторы
11:      String();
12:      String(const char *const);
13:      String(const String &);
14:      ~String();
15:
16:      // перегруженные операторы
17:      char & operator[](unsigned short offset);
18:      char operator[](unsigned short offset) const;
19:      String operator+(const String&);
20:      void operator+=(const String&);
21:      String & operator= (const String &);
22:
23:      // общие методы доступа
24:      unsigned short  GetLen() const { return itsLen; }
25:      const char * GetString() const { return itsString; }
26:
27:    private:
28:      String (unsigned short);         // закрытый конструктор
29:      char * itsString;
30:      unsigned short itsLen;
31:  };
32:
33:  // стандартный конструктор создает строку нулевой длины
34:  String::String()
35:  {
36:    itsString = new char[1];
```

```
37:    itsString[0] = '\0';
38:    itsLen=0;
39:  }
40:
41:  // закрытый (вспомогательный) конструктор,
42:  // используемый только методами класса для создания
43:  // строк необходимой длины, заполненных символом null.
44:  String::String(unsigned short len)
45:  {
46:    itsString = new char[len+1];
47:    for (unsigned short i=0; i<=len; i++)
48:      itsString[i] = '\0';
49:    itsLen=len;
50:  }
51:
52:  // Преобразует символьный массив в строку
53:  String::String(const char * const cString)
54:  {
55:    itsLen = strlen(cString);
56:    itsString = new char[itsLen+1];
57:    for (unsigned short i=0; i<itsLen; i++)
58:      itsString[i] = cString[i];
59:    itsString[itsLen]='\0';
60:  }
61:
62:  // конструктор копий
63:  String::String (const String & rhs)
64:  {
65:    itsLen=rhs.GetLen();
66:    itsString = new char[itsLen+1];
67:    for (unsigned short i=0; i<itsLen; i++)
68:      itsString[i] = rhs[i];
69:    itsString[itsLen] = '\0';
70:  }
71:
72:  // деструктор, освобождает выделенную память
73:  String::~String ()
74:  {
75:    delete [] itsString;
76:    itsLen = 0;
77:  }
78:
79:  // оператор присвоения, освобождает существующую память,
80:  // а затем копирует строку и ее размер
81:  String& String::operator=(const String & rhs)
82:  {
83:    if (this == &rhs)
84:      return *this;
85:    delete [] itsString;
86:    itsLen=rhs.GetLen();
87:    itsString = new char[itsLen+1];
88:    for (unsigned short i=0; i<itsLen; i++)
89:      itsString[i] = rhs[i];
90:    itsString[itsLen] = '\0';
91:    return *this;
92:  }
93:
94:  // непостоянный оператор индексирования, возвращает
```

```
 95:  // ссылку на символ, так что ее можно
 96:  // изменить
 97:  char & String::operator[](unsigned short offset)
 98:  {
 99:    if (offset > itsLen)
100:      return itsString[itsLen-1];
101:    else
102:      return itsString[offset];
103:  }
104:
105:  // постоянный оператор индексирования для использования
106:  // с постоянными объектами (см. конструктор копий)
107:  char String::operator[](unsigned short offset) const
108:  {
109:    if (offset > itsLen)
110:      return itsString[itsLen-1];
111:    else
112:      return itsString[offset];
113:  }
114:
115:  // создает новую строку, добавляя текущую
116:  // строку к rhs
117:  String String::operator+(const String& rhs)
118:  {
119:    unsigned short totalLen = itsLen + rhs.GetLen();
120:    String temp(totalLen);
121:    unsigned short i;
122:    for (i= 0; i<itsLen; i++)
123:      temp[i] = itsString[i];
124:    for (unsigned short j=0; j<rhs.GetLen(); j++, i++)
125:      temp[i] = rhs[j];
126:    temp[totalLen]='\0';
127:    return temp;
128:  }
129:
130:  // изменяет текущую строку, ничего не возвращая
131:  void String::operator+=(const String& rhs)
132:  {
133:    unsigned short rhsLen = rhs.GetLen();
134:    unsigned short totalLen = itsLen + rhsLen;
135:    String temp(totalLen);
136:    unsigned short i;
137:    for (i=0; i<itsLen; i++)
138:      temp[i] = itsString[i];
139:    for (unsigned short j=0; j<rhs.GetLen(); j++, i++)
140:      temp[i] = rhs[i-itsLen];
141:    temp[totalLen]='\0';
142:    *this = temp;
143:  }
144:
145:  int main()
146:  {
147:    String s1("initial test");
148:    cout << "S1:\t" << s1.GetString() << endl;
149:
150:    char * temp = "Hello World";
151:    s1 = temp;
152:    cout << "S1:\t" << s1.GetString() << endl;
```

```
153:
154:     char tempTwo[20];
155:     strcpy(tempTwo,"; nice to be here!");
156:     s1 += tempTwo;
157:     cout << "tempTwo:\t" << tempTwo << endl;
158:     cout << "S1:\t" << s1.GetString() << endl;
159:
160:     cout << "S1[4]:\t" << s1[4] << endl;
161:     s1[4]='x';
162:     cout << "S1:\t" << s1.GetString() << endl;
163:
164:     cout << "S1[999]:\t" << s1[999] << endl;
165:
166:     String s2(" Another string");
167:     String s3;
168:     s3 = s1+s2;
169:     cout << "S3:\t" << s3.GetString() << endl;
170:
171:     String s4;
172:     s4 = "Why does this work?";
173:     cout << "S4:\t" << s4.GetString() << endl;
174:     return 0;
175: }
```

Результат

```
S1:     initial test
S1:     Hello World
tempTwo:        ; nice to be here!
S1:     Hello World; nice to be here!
S1[4]:  o
S1:     Hellx World; nice to be here!
S1[999]:        !
S3:     Hellx World; nice to be here! Another string
S4:     Why does this work?
```

Анализ

Простой класс `String` объявлен в строках 7–31. Строки 11–13 содержат объявления трех конструкторов: стандартного, конструктора копий и конструктора, которому передают строку стиля C, завершающуюся символом `null`.

Чтобы облегчить пользователю работу со строками, класс `String` перегружает несколько операторов, включая оператор индекса (`[]`), оператор плюс (`+`) и оператор присвоения с суммой (`+=`). Оператор индекса перегружен дважды: один раз — как постоянная функция, возвращающая значение типа `char`, и второй раз — как непостоянная функция, возвращающая ссылку на значение типа `char`.

Непостоянная версия используется в таких операторах, как `S1[4]='x';` (строка 161). Это обеспечивает прямой доступ к любому символу строки. Получив таким образом ссылку на символ и вызвав эту функцию, можно изменить его значение.

Постоянная версия оператора используется в тех случаях, когда необходимо получить доступ к постоянному объекту класса `String`, например, в реализации конструктора копий (строка 63). Обратите внимание, что `rhs[i]` доступен, хотя `rhs` был объявлен как `const String &`. К этому объекту невозможно получить доступ, используя непостоянные функции-члены. Следовательно, оператор индекса необходимо перегрузить как постоянный.

Если возвращаемый объект окажется слишком большим, придется вернуть не сам объект, а постоянную ссылку на него. Но поскольку один символ занимает только один байт, нет смысла так поступать.

Стандартный конструктор реализован в строках 34–39. Он создает строку нулевой длины. Длина строки в классе String измеряется без учета концевого символа null. Таким образом, строка, созданная по умолчанию, содержит лишь концевой символ null.

Конструктор копий реализован в строках 63–70. Он устанавливает длину новой строки на единицу больше исходной (дополнительный символ необходим для завершающего null), а затем копирует каждый символ исходной строки во вновь созданную и завершает ее символом null.

В строках 53–60 реализован конструктор, которому передают исходную строку в стиле С (с концевым null). Этот конструктор подобен конструктору копий. Длина исходной строки вычисляется с помощью функции strlen() из стандартной библиотеки String.

В строке 28 как закрытая функция-член объявлен еще один конструктор, String(unsigned short). Таково было намерение автора, чтобы ни один из клиентов этого класса не мог создать строку произвольной длины. Этот конструктор создает строки для внутреннего использования, такие, как, например, operator+= в строке 131. Более подробная информация об операторе operator+= приведена далее в этой главе.

Конструктор String(unsigned short) заполняет все элементы строки нулевым символом ('\0'). Поэтому условием выхода из цикла for будет i<=len, а не i<len.

Деструктор, реализованный в строках 73–77, освобождает память, занимаемую строкой объекта класса. Удостоверьтесь, что при вызове оператора delete[] не забыты квадратные скобки, иначе вместо всего массива будет удален лишь первый его элемент.

Сначала оператор присвоения проверяет, не равна ли строка справа от оператора строке слева. Если это не так, то, удалив прежнюю строку, он создает новую и копирует в нее исходную. В качестве результата возвращается ссылка на новую строку, что позволяет осуществить присвоение типа:

```
String1 = String2 = String3;
```

Оператор индекса перегружен дважды (в строках 97–103 и 107–113). И в обоих случаях осуществляется проверка пределов массивов. Если пользователь попытается получить доступ к символу вне пределов массива, то последним возвращаемым символом окажется последний символ массива, т.е. len-1 элемент.

В строках 117–127 оператор плюс (+) перегружается в оператор конкатенации. Было бы очень удобно иметь возможность осуществлять конкатенацию строк аналогично простому сложению:

```
String3 = String1 + String2;
```

Для реализации этой возможности функция, заменяющая оператор плюс (+), вычисляет суммарную длину обеих строк и на основании результата создает временную строку temp. Для этого применяется закрытый конструктор, который получает целое число и создает строку, заполненную пустыми символами (null). Впоследствии эти пустые символы будут заменены содержимым двух исходных строк. Сначала во временную строку копируется левая исходная строка (*this), затем — правая (rhs).

Оператор суммы (строка 127) возвращает временную строку как значение, которое присваивается строке, расположенной слева от оператора (string1). Оператор += (строки 131–143) работает с уже существующими строками, находящимися слева от оператора string1 += string2. Он работает аналогично оператору суммы, за исключением того, что временное значение присваивается текущей строке *this = temp (строка 142).

Функция main() (строки 145–175) проверяет работоспособность созданного класса. В строке 147 создается объект класса String с помощью конструктора, получающего строку в стиле С с пустым символом в конце. Строка 148 выводит на экран ее содержимое, используя функцию доступа GetString(). В строке 150 создается другая

строка стиля C. Строка 151 проверяет оператор присвоения, а строка 152 выводит на экран результаты.

В строке 154 создается третья строка стиля C — `tempTwo`. В строке 155 вызов функции `strcpy()` заполняет буфер символами "`; nice to be here!`". Строка 156 вызывает оператор `+=` и добавляет к строке `s1` строку `tempTwo`. Строка 158 выводит результаты на экран.

В строке 160 возвращается и выводится на экран пятый символ строки `s1`. В строке 161 с помощью оператора индекса (`[]`) ему присваивается новое значение, а в строке 162 на экран выводится результат, демонстрирующий факт внесения изменений.

В строке 164 предпринята попытка доступа к символу за пределами массива, но возвращен был лишь последний символ массива, как и было задумано.

В строках 166 и 167 создаются еще два объекта класса `String`, а в строке 168 происходит их сложение. Строка 169 выводит результаты на экран.

В строке 171 создается новый объект класса `String` — `s4`, вызов оператора присвоения расположен в строке 172, а строка 173 выводит результат на экран. Можно было бы засомневаться: "В строке 21 оператор присвоения определен так, чтобы получать постоянную ссылку на объект класса `String`, а здесь передается строка в стиле C. Допустимо ли это?"

Хоть компилятор и ожидает объект класса `String`, но, получив символьный массив, он проверяет, можно ли преобразовать его в строку. А в строке 12 как раз объявлен конструктор, который создает объекты класса `String` из символьных массивов. Компилятор создает из символьного массива временную строку и передает ее оператору присвоения. Этот процесс известен под названием *неявное приведение* (implicit casting, или promotion). Если бы не был объявлен соответствующий конструктор для реализации подобной функции, такое присвоение привело бы к ошибке компиляции.

Как можно заметить, просмотрев листинг 13.14, созданный класс `String` получился вполне работоспособным. Его код, правда, достаточно велик, но, к счастью, стандартная библиотека C++ предоставляет даже более удобный класс `String`, которым можно воспользоваться, подключив библиотеку `<string>`.

Связанные списки и другие структуры

Массивы напоминают лоток для яиц. Это прекрасный контейнер фиксированного размера. Если контейнер слишком велик, то избыток пространства будет растрачен впустую. Если контейнер мал, его не хватит для размещения содержимого.

Один из способов решения этой проблемы продемонстрирован в листинге 13.9. Но при использовании больших массивов или при перемещении, удалении и вставке в массив элементов большое количество операций выделения и освобождения памяти может оказаться весьма накладным.

Еще одним решением этой проблемы являются связанные списки. *Связанный список* — это структура данных, состоящая из небольших контейнеров, способных поддерживать связь с аналогичными контейнерами. Основная мысль состоит в том, чтобы создать класс, который содержит один объект пользовательского типа (например, `Cat` или `Rectangle`) и указатель на следующий контейнер. Таким образом, созданный контейнер способен хранить один объект необходимого типа, а также поддерживать связь со следующим аналогичным контейнером, что позволяет построить из них цепочку необходимой длины.

Связанные списки — это отдельная тема для рассмотрения. Более подробная информация о них содержится в приложении Д, "Связанные списки".

Классы массивов

Использование собственного класса массива вместо стандартного встроенного массива предоставляет множество преимуществ. Во-первых, можно предотвратить возможность записи за пределами массива. Во-вторых, можно создать такой класс массива, размер которого будет изменяться динамически: при создании он будет размером в один элемент, а по мере добавления новых приобретет необходимый размер.

Элементы массива можно автоматически сортировать и выстраивать в необходимом порядке. Кроме того, такой подход позволяет создать целый ряд специальных типов массивов. Наиболее популярными из них являются:

- **коллекция** (collection) — элементы упорядочены и отсортированы в определенном порядке;
- **набор** (set) — ни один из элементов не повторяется;
- **словарь** (dictionary) — набор соответствующих друг другу пар элементов, в которых значение одного из них можно получить из значения другого;
- **разреженный массив** (sparse array) — разрешены индексы любой величины, но память занимают только те значения, которые содержатся в массиве; таким образом, можно обратиться и к элементу `SparseArray[5]`, и к элементу `SparseArray[200]`, но память будет выделена только для пятого элемента;
- **мешок** (bag), или **мультимножество**, — неупорядоченный набор элементов, добавлять и обращаться к которым можно в произвольном порядке.

Перегрузив оператор индексирования (`[]`), можно преобразовать связанный список в коллекцию. Исключив дублирование элементов, можно преобразовать коллекцию в набор. Если каждый объект списка имеет пару значений, его можно использовать для создания словаря или разреженного массива.

> Хотя свой собственный класс массива и предоставляет множество преимуществ, использование стандартных библиотечных реализаций подобных классов, как правило, все же удобнее.

Резюме

Сегодня рассматривалось создание массивов в языке C++. *Массив* — это набор строго фиксированного количества однотипных элементов.

Массивы не проверяют свой размер. Поэтому возможна ситуация, когда элемент записывается за пределами области памяти, выделенной для массива. Обычно это приводит к катастрофе. Массивы начинаются с `0`. Частой ошибкой является попытка записи элемента номер *n* в массив из *n* элементов.

Массивы могут быть одно- или многомерными. В любом случае элементы массива могут быть инициализированы при создании, вне зависимости от того, содержит ли он встроенные типы (такие, как `int`) или объекты класса, располагающего стандартным конструктором.

Массивы и их содержимое могут быть размещены как в динамической памяти, так и в стеке. При удалении массивов, размещенных в динамической памяти, не забывайте использовать квадратные скобки в операторе `delete[]`.

Имя массива представляет собой постоянный указатель на его первый элемент. Указатели и массивы используют арифметические операции над указателями для поиска необходимого элемента массива.

Строки — это разновидность массива символов. Язык C++ обладает специальными средствами для работы с символьными массивами, включая возможность инициализировать их с помощью строки, заключенной в парные кавычки.

Вопросы и ответы

- **Что находится в неинициализированном элементе массива?**

  Любое значение, находившееся в этой области памяти ранее. Результаты использования неинициализированного элемента непредсказуемы. Если компилятор соответствует стандарту C++, то элементы массива, не являющиеся статическими или локальными объектами, будут инициализированы нулем.

- **Можно ли объединять массивы?**

  Да. Чтобы объединить массивы, необходимо использовать указатели. Объединять строки еще проще: для них можно использовать встроенные функции, например, `strcat()`.

- **Зачем создавать связанный список, если можно использовать массив?**

  Массив имеет фиксированный размер, в то время как связанный список способен динамически изменять свой размер в процессе выполнения программы. Более подробная информация о связанных списках приведена в приложении Д, “Связанные списки”.

- **Зачем использовать встроенные массивы, если класс массива работает лучше?**

  Применение встроенных массивов проще и быстрее.

- **Должен ли класс `String` использовать указатель `char *`, чтобы хранить содержимое строки?**

  Нет. Для хранения можно использовать любой тип памяти. Выбирать нужно тот, который подходит наиболее.

Коллоквиум

В этом разделе предлагаются вопросы для самоконтроля и закрепления полученных знаний, а также упражнения, которые помогут применить изученное на практике. Попытайтесь самостоятельно ответить на вопросы и выполнить все задания и только потом сверьте полученные результаты с ответами в приложении Г, “Ответы”. Не приступайте к изучению материала следующей главы, если остались неясными хотя бы некоторые из предложенных ниже вопросов.

Контрольные вопросы

1. Какой из элементов массива `SomeArray[25]` будет первым, а какой — последним?
2. Как объявить многомерный массив?
3. Инициализируйте элементы массива из вопроса 2.
4. Сколько элементов содержит массив `SomeArray[10][5][20]`?
5. Чем связанный список отличается от массива?
6. Сколько символов хранится в строке “`Jesse knows C++`”?
7. Каким будет последний символ в строке “`Brad is a nice guy`”?

Упражнения

1. Объявите двумерный массив, представляющий поле для игры в крестики-нолики.
2. Напишите код, инициализирующий значением 0 все элементы массива упражнения 1.
3. Напишите программу, которая содержит четыре массива. Три первых массива должны содержать имена, инициалы и фамилии. Используя изученную на сегодняшнем уроке функцию копирования строк, организуйте конкатенацию и копирование их строк в четвертый массив, содержащий полные имена.
4. **Отладка.** Найдите ошибку в следующем фрагменте кода:

```
unsigned short SomeArray[5][4];
for (int i=0; i<4; i++)
  for (int j=0; j<5; j++)
    SomeArray[i][j] = i+j;
```

5. **Отладка.** Найдите ошибку в следующем фрагменте кода:

```
unsigned short SomeArray[5][4];
for (int i=0; i<=5; i++)
  for (int j=0; j<=4; j++)
    SomeArray[i][j] = 0;
```