Предположим, что вы разработали компьютерную игру под названием "Враждебный пользователь", в которой игроки состязаются с замысловатым и недружественным компьютерным интерфейсом. Теперь вам необходимо написать программу, которая отслеживает ежемесячные объемы продаж этой игры в течение пятилетнего периода. Или, скажем, вам нужно провести инвентаризацию торговых карт героев-хакеров. Очень скоро вы придете к выводу, что для накопления и обработки информации вам требуется нечто большее, чем простые базовые типы С++. И С++ предлагает это нечто большее, а именно — составные типы. Это типы, состоящие из базовых целочисленных типов и типов с плавающей точкой. Наиболее развитым составным типом является класс — оплот объектно-ориентированного программирования (ООП), к которому мы стремимся двигаться. Но С++ также поддерживает несколько более скромных составных типов, которые взяты из языка С. Массив, например, может хранить множество значений одного и того же типа. Отдельный вид массива может хранить строки, которые являются последовательностями символов. Структуры могут хранить по нескольку значений разных типов. Кроме того, есть еще указатели, которые представляют собой переменные, сообщающие компьютеру местонахождение данных в памяти. Все эти составные формы данных (кроме классов) мы рассмотрим в настоящей главе. Вы кратко ознакомитесь с операциями `new` и `delete`, а также получите первое представление о классе С++ по имени `string`, который предлагает альтернативный способ работы со строками.

Введение в массивы

Массив — это структура данных, которая содержит множество значений, относящихся к одному и тому же типу. Например, массив может содержать 60 значений типа `int`, которые представляют информацию об объемах продаж за 5 лет, 12 значений типа `short`, представляющих количество дней в каждом месяце, или 365 значений типа `float`, которые указывают ежедневные расходы на питание в течение года. Каждое значение сохраняется в отдельном элементе массива, и компьютер хранит все элементы массива в памяти последовательно — друг за другом.

Для создания массива используется оператор объявления. Объявление массива должно описывать три аспекта:

- тип значений каждого элемента;
- имя массива;
- количество элементов в массиве.

В С++ это достигается модификацией объявления простой переменной, к которому добавляются квадратные скобки, содержащие внутри количество элементов. Например, следующее объявление создает массив по имени `months`, имеющий 12 элементов, каждый из которых может хранить одно значение типа `short`:

```
short months[12];    // создает массив из 12 элементов типа short
```

В сущности, каждый элемент — это переменная, которую можно трактовать как простую переменную.

Так выглядит общая форма объявления массива:

```
имяТипа имяМассива[размерМассива];
```

Выражение *размерМассива*, представляющее количество элементов, должно быть целочисленной константой, такой как 10, значением `const` либо константным выражением вроде `8 * sizeof(int)`, в котором все значения известны на момент компи-

ляции. В частности, *размерМассива* не может быть переменной, значение которой устанавливается во время выполнения программы. Однако позднее в этой главе вы узнаете, как с использованием операции new обойти это ограничение.

Массив как составной тип

Массив называют *составным типом*, потому что он строится из какого-то другого типа. (В языке C используется термин *производный тип*, но поскольку понятие *производный* в C++ применяется для описания отношений между классами, пришлось ввести новый термин.) Вы не можете просто объявить, что нечто является массивом; это всегда должен быть массив элементов конкретного типа. Обобщенного типа массива не существует. Вместо этого имеется множество специфических типов массивов, таких как массив char или массив long. Например, рассмотрим следующее объявление:

```
float loans[20];
```

Типом переменной loans будет не просто "массив", а "массив float". Это подчеркивает, что массив loans построен из типа float.

Большая часть пользы от массивов определяется тем фактом, что к его элементам можно обращаться индивидуально. Способ, который позволяет это делать, заключается в использовании *индекса* для нумерации элементов. Нумерация массивов в C++ начинается с нуля. (Это является обязательным — вы должны начинать с нуля. Это особенно важно запомнить программистам, ранее работавшим на языках Pascal и BASIC.) Для указания элемента массива в C++ используется обозначение с квадратными скобками и индексом между ними. Например, months[0] — это первый элемент массива months, а months[11] — его последний элемент. Обратите внимание, что индекс последнего элемента на единицу меньше, чем размер массива (рис. 4.1). Таким образом, объявление массива позволяет создавать множество переменных в одном объявлении, и вы затем можете использовать индекс для идентификации и доступа к индивидуальным элементам.

Важность указания правильных значений индекса

Компилятор не проверяет правильность указываемого индекса. Например, компилятор не станет жаловаться, если вы присвоите значение несуществующему элементу months[101]. Однако такое присваивание может вызвать проблемы во время выполнения программы — возможно, повреждение данных или кода, а может быть и аварийное завершение программы. То есть обеспечение правильности значений индекса возлагается на программиста.

Массив, хранящий семь значений, каждое из которых является переменной типа int

Рис. 4.1. Создание массива

Небольшая программа анализа, представленная в листинге 4.1, демонстрирует несколько свойств массивов, включая их объявление, присваивание значение его элементам, а также инициализацию.

Листинг 4.1. arrayone.cpp

```
// arrayone.cpp -- небольшие массивы целых чисел
#include <iostream>
int main()
{
    using namespace std;
    int yams[3];                 // создание массива из трех элементов
    yams[0] = 7;                 // присваивание значения первому элементу
    yams[1] = 8;
    yams[2] = 6;

    int yamcosts[3] = {20, 30, 5};  // создание и инициализация массива
    // Примечание. Если ваш компилятор C++ не может инициализировать
    // этот массив, используйте static int yamcosts[3] вместо int yamcosts[3]

    cout << "Total yams = ";
    cout << yams[0] + yams[1] + yams[2] << endl;
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";
    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";

    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
    return 0;
}
```

Ниже показан вывод программы из листинга 4.1:

```
Total yams = 21
The package with 8 yams costs 30 cents per yam.
The total yam expense is 410 cents.

Size of yams array = 12 bytes.
Size of one element = 4 bytes.
```

Замечания по программе

Сначала программа в листинге 4.1 создает массив из трех элементов по имени yams. Поскольку yams имеет три элемента, они нумеруются от 0 до 2, и arrayone.cpp использует значения индекса от 0 до 2 для присваивания значений трем отдельным элементам. Каждый индивидуальный элемент yams — это переменная типа int, со всеми правилами и привилегиями типа int, поэтому arrayone.cpp может (и делает это) присваивать значения элементам, складывать элементы, перемножать их и отображать.

В этой программе применяется длинный способ присваивания значений элементам yams. C++ также позволяет инициализировать элементы массива непосредственно в операторе объявления. В листинге 4.1 демонстрируется этот сокращенный способ при установке значений элементов массива yamcosts:

```
int yamcosts[3] = {20, 30, 5};
```

Он просто предоставляет разделенный запятыми список значений (*список инициализации*), заключенный в фигурные скобки. Пробелы в списке не обязательны. Если вы не инициализируете массив, объявленный внутри функции, его элементы остаются неопределенными. Это значит, что элементы получают случайные значения, которые зависят от предыдущего содержимого области памяти, выделенной для такого массива.

Далее программа использует значения массива в нескольких вычислениях. Эта часть программы выглядит несколько беспорядочно со всеми этими индексами и скобками. Цикл for, который будет описан в главе 5, предоставляет мощный способ работы с массивами и исключает необходимость явного указания индексов. Но пока мы ограничимся небольшими массивами.

Как вы, возможно, помните, операция sizeof возвращает размер в байтах типа или объекта данных. Обратите внимание, что применение sizeof к имени массива дает количество байт, занимаемых всем массивом. Однако использование sizeof в отношении элемента массива дает размер в байтах одного элемента. Это иллюстрирует тот факт, что yams — массив, но yams[1] — просто int.

Правила инициализации массивов

В C++ существует несколько правил, касающихся инициализации массивов. Они ограничивают, когда вы можете ее осуществлять, и определяют, что случится, если количество элементов массива не соответствует количеству элементов инициализатора. Давайте рассмотрим эти правила.

Вы можете использовать инициализацию *только* при объявлении массива. Ее нельзя выполнить позже, и нельзя присваивать один массив другому:

```
int cards[4] = {3, 6, 8, 10};      // все в порядке
int hand[4];                       // все в порядке
hand[4] = {5, 6, 7, 9};            // не допускается
hand = cards;                      // не допускается
```

Однако можно использовать индексы и присваивать значения элементам массива индивидуально.

При инициализации массива можно указывать меньше значений, чем в массиве объявлено элементов. Например, следующий оператор инициализирует только первые два элемента массива hotelTips:

```
float hotelTips[5] = {5.0, 2.5};
```

Если вы инициализируете массив частично, то компилятор присваивает остальным элементом нулевые значения. Это значит, что инициализировать весь массив нулями очень легко — для этого просто нужно явно инициализировать нулем его первый элемент, а инициализацию остальных элементов поручить компилятору:

```
long totals[500] = {0};
```

Следует отметить, что в случае инициализации массива с применением {1} вместо {0} только первый элемент будет установлен в 1; остальные по-прежнему получат значение 0.

Если при инициализации массива оставить квадратные скобки пустыми, то компилятор C++ самостоятельно пересчитает элементы. Предположим, например, что есть следующее объявление:

```
short things[] = {1, 5, 3, 8};
```

Компилятор сделает things массивом из пяти элементов.

Позволять ли компилятору самостоятельно подсчитывать элементы?

Часто компилятор при подсчете элементов получает не то количество, которое, как вы ожидаете, должно быть. Причиной может быть, например, непреднамеренный пропуск одного или нескольких значений в списке инициализации. Однако, как вы вскоре убедитесь, такой подход может быть вполне безопасным для инициализации символьного массива строкой. И если главная цель состоит в том, чтобы программа, а не вы, знала размер массива, то можно записать примерно следующий код:

```
short things[] = {1, 5, 3, 8};
int num_elements = sizeof things / sizeof (short);
```

Удобно это или нет — зависит от сложившихся обстоятельств.

Инициализация массивов в C++11

Как упоминалось в главе 3, в C++11 форма инициализации с фигурными скобками (списковая инициализация) стала универсальной для всех типов. Массивы уже используют списковую инициализацию, но в версии C++11 появились дополнительные возможности.

Во-первых, при инициализации массива можно отбросить знак =:

```
double earnings[4] {1.2e4, 1.6e4, 1.1e4, 1.7e4}; // допускается в C++11
```

Во-вторых, можно использовать пустые фигурные скобки для установки всех элементов в 0:

```
unsigned int counts[10] = {};          // все элементы устанавливаются в 0
float balances[100] {};                // все элементы устанавливаются в 0
```

В-третьих, как обсуждалось в главе 3, списковая инициализация защищает от сужения:

```
long plifs[] = {25, 92, 3.0};                  // не разрешено
char slifs[4] {'h', 'i', 1122011, '\0'};  // не разрешено
char tlifs[4] {'h', 'i', 112, '\0'};      // разрешено
```

Первая инициализация не допускается, т.к. преобразование из типа с плавающей точкой в целочисленный тип является сужением, даже если значение с плавающей точкой содержит после десятичной точки только нули. Вторая инициализация не допускается, поскольку 1122011 выходит за пределы диапазона значений типа char, предполагая, что char занимает 8 бит. Третья инициализация выполняется успешно, т.к. несмотря на то, что 112 является значением int, оно находится в рамках диапазона типа char.

Стандартная библиотека шаблонов C++ (STL) предлагает альтернативу массивам — шаблонный класс vector, а в C++11 еще добавлен шаблонный класс array. Эти альтернативы являются более сложными и гибкими, нежели встроенный составной тип массива. Они кратко будут рассматриваться далее в этой главе и более подробно — в главе 16.

Строки

Строка — это серия символов, сохраненная в расположенных последовательно байтах памяти. В C++ доступны два способа работы со строками. Первый, унаследованный от C и часто называемый *строками в стиле C*, рассматривается в настоящей главе сначала. Позже будет описан альтернативный способ, основанный на библиотечном классе string.

Идея серии символов, сохраняемых в последовательных байтах, предполагает хранение строки в массиве char, где каждый элемент содержится в отдельном элементе массива. Строки предоставляют удобный способ хранения текстовой информации, такой как сообщения для пользователя или его ответы. Строки в стиле C обладают специальной характеристикой: последним в каждой такой строке является *нулевой символ*. Этот символ, записываемый как \0, представляет собой символ с ASCII-кодом 0, который служит меткой конца строки. Например, рассмотрим два следующих объявления:

```
char dog[8] = { 'b', 'e', 'a', 'u', 'x', ' ', 'I', 'I'};    // это не строка
char cat[8] = {'f', 'a', 't', 'e', 's', 's', 'a', '\0'};    // а это — строка
```

Обе эти переменные представляют собой массивы char, но только вторая из них является строкой. Нулевой символ играет фундаментальную роль в строках стиля C. Например, в C++ имеется множество функций для обработки строк, включая те, что используются cout. Все они обрабатывают строки символ за символом до тех пор, пока не встретится нулевой символ. Если вы просите объект cout отобразить такую строку, как cat из предыдущего примера, он выводит первых семь символов, обнаруживает нулевой символ и на этом останавливается. Однако если вы вдруг решите вывести в cout массив dog из предыдущего примера, который не является строкой, то cout напечатает восемь символов из этого массива и будет продолжать двигаться по памяти, байт за байтом, интерпретируя каждый из них как символ, подлежащий выводу, пока не встретит нулевой символ. Поскольку нулевые символы, которые, по сути, представляют собой байты, содержащие нули, встречаются в памяти довольно часто, ошибка обычно обнаруживается быстро, но в любом случае вы не должны трактовать нестроковые символьные массивы как строки.

Пример инициализации массива cat выглядит довольно громоздким и утомительным — множество одиночных кавычек плюс необходимость помнить о нулевом символе. Не волнуйтесь. Существует более простой способ инициализации массива с помощью строки. Для этого просто используйте строку в двойных кавычках, которая называется *строковой константой* или *строковым литералом*, как показано ниже:

```
char bird[11] = "Mr. Cheeps";    // наличие символа \0 подразумевается
char fish[] = "Bubbles";         // позволяет компилятору подсчитать
                                 // количество элементов
```

Строки в двойных кавычках всегда неявно включают ограничивающий нулевой символ, поэтому указывать его явно не требуется (рис. 4.2.) К тому же разнообразные средства ввода C++, предназначенные для чтения строки с клавиатурного ввода в массив char, автоматически добавляют завершающий нулевой символ. (Если при компиляции программы из листинга 4.1 вы обнаружите необходимость в использовании ключевого слова static для инициализации массива, это также понадобится сделать с показанными выше массивами char.)

char boss[8] = "Bozo";

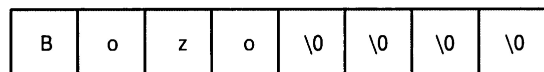


Рис. 4.2. Инициализация массива строкой

Разумеется, вы должны обеспечить достаточный размер массива, чтобы в него поместились все символы строки, включая нулевой. Инициализация символьного массива строковой константой — это один из тех случаев, когда безопаснее поручить компилятору подсчет количества элементов в массиве. Если сделать массив больше строки, никаких проблем не возникнет — только непроизводительный расход пространства. Причина в том, что функции, которые работают со строками, руководствуются позицией нулевого символа, а не размером массива. В C++ не накладывается никаких ограничений на длину строки.

На заметку!

При определении минимального размера массива, необходимого для хранения строки, не забудьте учесть при подсчете завершающий нулевой символ.

Обратите внимание, что строковая константа (в двойных кавычках) не взаимозаменяема с символьной константой (в одинарных кавычках). Символьная константа, такая как `'S'`, представляет собой сокращенное обозначение для кода символа. В системе ASCII константа `'S'` — это просто другой способ записи кода 83. Поэтому следующий оператор присваивает значение 83 переменной `shirt_size`:

```
char shirt_size = 'S';    // нормально
```

С другой стороны, `"S"` не является символьной константой; это строка, состоящая из двух символов — `S` и `\0`. Хуже того, `"S"` в действительности представляет адрес памяти, по которому размещается строка. Это значит, что приведенный ниже оператор означает попытку присвоить адрес памяти переменной `shirt_size`:

```
char shirt_size = "S";    // не допускается по причине несоответствия типов
```

Поскольку адрес памяти — это отдельный тип в C++, компилятор не пропустит подобную бессмыслицу. (Мы вернемся к этому моменту позже в данной главе, во время рассмотрения указателей.)

Конкатенация строковых литералов

Иногда строки могут оказаться слишком большими, чтобы удобно разместиться в одной строке кода. C++ позволяет выполнять конкатенацию строковых литералов — т.е. комбинировать две строки с двойными кавычками в одну. В действительности любые две строковые константы, разделенные только пробельным символом (пробелами, символами табуляции и символами новой строки), автоматически объединяются в одну. Таким образом, следующие три оператора вывода эквивалентны:

```
cout << "I'd give my right arm to be" " a great violinist.\n";
cout << "I'd give my right arm to be a great violinist.\n";
cout << "I'd give my right ar"
"m to be a great violinist.\n";
```

Обратите внимание, что такие объединения не добавляют никаких пробелов к объединяемым строкам. Первый символ второй строки немедленно следует за последним символом первой, не считая `\0` в первой строке. Символ `\0` из первой строки заменяется первым символом второй строки.

Использование строк в массивах

Два наиболее распространенных метода помещения строки в массив заключаются в инициализации массива строковой константой и чтением из клавиатурного или файлового ввода в массив. В листинге 4.2 эти подходы демонстрируются за счет ини-

циализации одного массива строкой в двойных кавычках и использования cin для помещения вводимой строки в другой массив. В программе также применяется стандартная библиотечная функция stlen() для получения длины строки. Стандартный заголовочный файл cstring (или string.h в более старых реализациях) предоставляет объявления для этой и многих других функций, работающих со строками.

Листинг 4.2. strings.cpp

```
// strings.cpp -- сохранение строк в массиве
#include <iostream>
#include <cstring>                    // для функции strlen()
int main()
{
    using namespace std;
    const int Size = 15;
    char name1[Size];                 // пустой массив
    char name2[Size] = "C++owboy";    // инициализация массива
    // ПРИМЕЧАНИЕ: некоторые реализации могут потребовать
    // ключевого слова static для инициализации массива name2

    cout << "Howdy! I'm " << name2;
    cout << "! What's your name?\n";
    cin >> name1;
    cout << "Well, " << name1 << ", your name has ";
    cout << strlen(name1) << " letters and is stored\n";
    cout << "in an array of " << sizeof(name1) << " bytes.\n";
    cout << "Your initial is " << name1[0] << ".\n";
    name2[3] = '\0';                   // установка нулевого символа
    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << endl;
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.2:

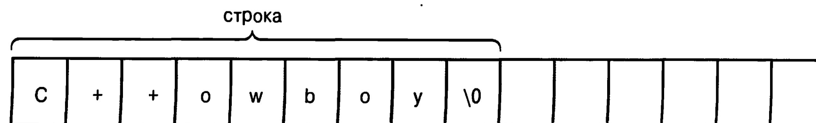
```
Howdy! I'm C++owboy! What's your name?
Basicman
Well, Basicman, your name has 8 letters and is stored
in an array of 15 bytes.
Your initial is B.
Here are the first 3 characters of my name: C++
```

Замечания по программе

Чему учит код в листинге 4.2? Первым делом, обратите внимание, что операция sizeof возвращает размер всего массива — 15 байт, но функция strlen() возвращает размер строки, хранящейся в массиве, а не размер самого массива. К тому же strlen() подсчитывает только видимые символы, без нулевого символа-ограничителя. То есть эта функция возвращает в качестве длины Basicman значение 8, а не 9. Если cosmic представляет собой строку, то минимальный размер массива для размещения этой строки вычисляется как strlen(cosmic) + 1.

Поскольку name1 и name2 — массивы, для доступа к отдельным символам в этих массивах можно использовать индексы. Например, в программе для поиска первого символа массива name1 применяется name1[0]. Кроме того, программа присваивает элементу name2[3] нулевой символ. Это завершает строку после трех символов, хотя в массиве остаются еще символы (рис. 4.3).

```
const int Size = 15;
char name2[Size] = "C++owboy";
```




```
name2[3] = '\0';
```

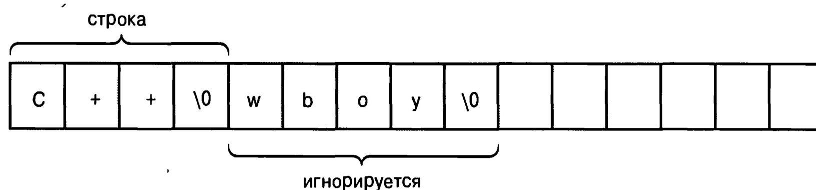


Рис. 4.3. Сокращение строки с помощью \0

Обратите внимание, что в программе из листинга 4.2 для указания размера массива используется символическая константа. Часто размер массива нужно указывать в нескольких операторах программы. Применение символических констант для представления размера массива упрощает внесение изменений, связанных с длиной массива; в таких случаях изменить размер потребуется только в одном месте — там, где определена символическая константа.

Риски, связанные с вводом строк

Программа `string.cpp` имеет недостаток, скрытый за часто используемой в литературе техникой тщательного выбора примеров ввода. В листинге 4.3 демонстрируется тот факт, что строковый ввод может оказаться непростым.

Листинг 4.3. insrt1.cpp

```
// insrt1.cpp -- чтение более одной строки
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";                    // запрос имени
    cin >> name;
    cout << "Enter your favorite dessert:\n";        // запрос любимого десерта
    cin >> dessert;
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Назначение программы из листинга 4.3 простое: прочитать имя пользователя и название его любимого десерта, введенные с клавиатуры, и затем отобразить эту информацию.

Ниже приведен пример запуска:

```
Enter your name:
Alistair Dreeb
Enter your favorite dessert:
I have some delicious Dreeb for you, Alistair.
```

Мы даже не получили возможности ответить на вопрос о десерте! Программа показала вопрос и затем немедленно перешла к отображению заключительной строки.

Проблема связана с тем, как cin определяет, когда ввод строки завершен. Вы не можете ввести нулевой символ с клавиатуры, поэтому cin требуется что-то другое для нахождения конца строки. Подход, принятый в cin, заключается в использовании пробельных символов для разделения строк — пробелов, знаков табуляции и символов новой строки. Это значит, что cin читает только одно слово, когда получает ввод для символьного массива. После чтения слова cin автоматически добавляет ограничивающий нулевой символ при помещении строки в массив.

Практический результат этого примера заключается в том, что cin читает слово Alistair как полную первую строку и помещает его в массив name. При этом второе слово, Dreeb, остается во входной очереди. Когда cin ищет ввод, отвечающий на вопрос о десерте, он находит там Dreeb. Затем cin захватывает слово Dreeb и помещает его в массив dessert (рис. 4.4).

Еще одна проблема, которая не была обнаружена в примере запуска, состоит в том, что вводимая строка, в свою очередь, может быть длиннее, чем целевой массив. При таком использовании cin, как это сделано в примере, нет никакой защиты от помещения 30-символьной строки в 20-символьный массив.

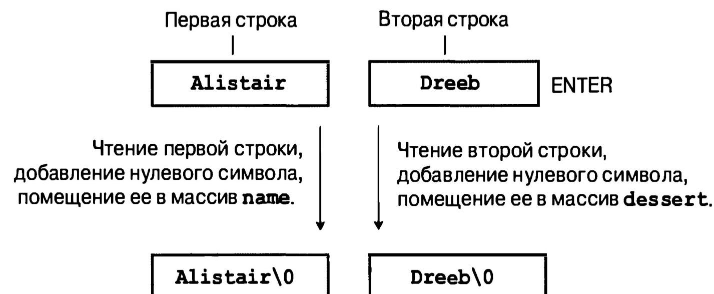


Рис. 4.4. *Строковый ввод с точки зрения* cin

Многие программы зависят от строкового ввода, поэтому нам стоит рассмотреть данную тему глубже. Некоторые из наиболее усовершенствованных средств cin будут подробно описаны в главе 17.

Построчное чтение ввода

Чтение строкового ввода по одному слову за раз — часто не является желательным поведением. Например, предположим, что программа запрашивает у пользователя ввод города, и пользователь отвечает вводом **New York** или **Sao Paulo**. Вы бы хотели, чтобы программа прочитала и сохранила полные названия, а не только New и Sao. Чтобы иметь возможность вводить целые фразы вместо отдельных слов, необходим другой подход к строковому вводу. Точнее говоря, нужен метод, ориентированный на строки, вместо метода, ориентированного на слова. К счастью, у класса istream, экземпляром которого является cin, есть функции-члены, предназначен-

ные для строчно-ориентированного ввода: getline() и get(). Оба читают полную строку ввода — т.е. вплоть до символа новой строки. Однако getline() затем отбрасывает символ новой строки, в то время как get() оставляет его во входной очереди. Давайте рассмотрим их детально, начиная с getline().

Строчно-ориентированный ввод с помощью getline()

Функция getline() читает целую строку, используя символ новой строки, который передан клавишей <Enter>, для обозначения конца ввода. Этот метод инициируется вызовом функции cin.getline(). Функция принимает два аргумента. Первый аргумент — это имя места назначения (т.е. массива, который сохраняет введенную строку), а второй — максимальное количество символов, подлежащих чтению. Если, скажем, установлен предел 20, то функция читает не более 19 символов, оставляя место для автоматически добавляемого в конец нулевого символа. Функция-член getline() прекращает чтение, когда достигает указанного предела количества символов или когда читает символ новой строки — смотря, что произойдет раньше.

Например, предположим, что вы хотите воспользоваться getline() для чтения имени в 20-элементный массив name. Для этого следует указать такой вызов:

```
cin.getline(name,20);
```

Он читает полную строку в массив name, предполагая, что строка состоит не более чем из 19 символов. (Функция-член getline() также принимает необязательный третий аргумент, который обсуждается в главе 17.)

В листинге 4.4 представлен модифицированный пример из листинга 4.3 с применением cin.getline() вместо простого cin. В остальном программа осталась прежней.

Листинг 4.4. insrt2.cpp

```
// insrt2.cpp -- чтение более одного слова с помощью getline
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";                  // запрос имени
    cin.getline(name, ArSize);                     // читать до символа новой строки
    cout << "Enter your favorite dessert:\n";      // запрос любимого десерта
    cin.getline(dessert, ArSize);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.4:

```
Enter your name:
Dirk Hammernose
Enter your favorite dessert:
Radish Torte
I have some delicious Radish Torte for you, Dirk Hammernose.
```

Теперь программа читает полное имя и название блюда. Функция `getline()` удобным образом принимает по одной строке за раз. Она читает ввод до новой символа строки, помечая конец строки, но не сохраняя при этом сам символ новой строки. Вместо этого она заменяет его нулевым символом при сохранении строки (рис. 4.5).

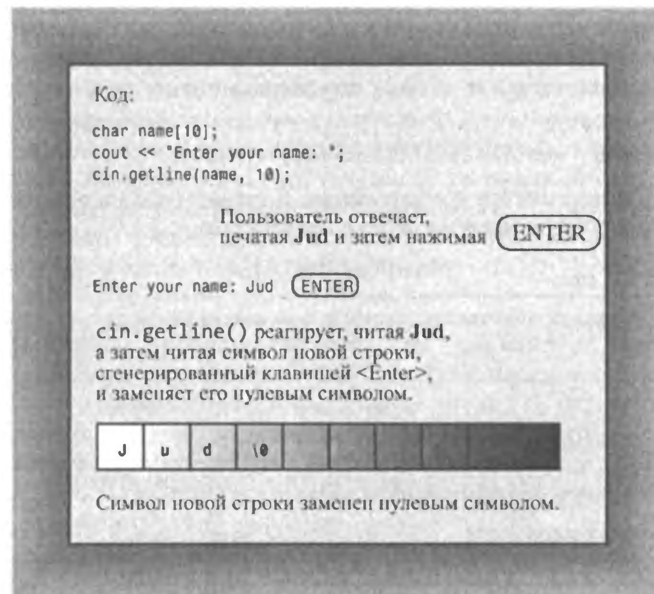


Рис. 4.5. *`getline()` читает и заменяет символ новой строки*

Строчно-ориентированный ввод с помощью `get()`

Теперь попробуем другой подход. Класс `istream` имеет функцию-член `get()`, которая доступна в различных вариантах. Один из них работает почти так же, как `getline()`. Он принимает те же аргументы, интерпретирует их аналогичным образом, и читает до конца строки. Но вместо того, чтобы прочитать и отбросить символ новой строки, `get()` оставляет его во входной очереди. Предположим, что используются два вызова `get()` подряд:

```
cin.get(name, ArSize);
cin.get(dessert, Arsize);     // проблема
```

Поскольку первый вызов оставляет символ новой строки во входной очереди, получается, что символ новой строки оказывается первым символом, который видит следующий вызов. Таким образом, второй вызов `get()` заключает, что он достиг конца строки, не найдя ничего интересного, что можно было бы прочитать. Без посторонней помощи `get()` вообще не может преодолеть этот символ новой строки.

К счастью, на помощь приходят различные варианты `get()`. Вызов `cin.get()` без аргументов читает одиночный следующий символ, даже если им будет символ новой строки, поэтому вы можете использовать его для того, чтобы отбросить символ новой строки и подготовиться к вводу следующей строки. То есть следующая последовательность будет работать правильно:

```
cin.get(name, ArSize);        // чтение первой строки
cin.get();                    // чтение символа новой строки
cin.get(dessert, Arsize);     // чтение второй строки
```

Другой способ применения `get()` состоит в *конкатенации*, или соединении, двух вызовов функций-членов класса, как показано в следующем примере:

```
cin.get(name, ArSize).get();    // конкатенация функций-членов
```

Такую возможность обеспечивает то, что `cin.get(name, ArSize)` возвращает объект `cin`, который затем используется в качестве объекта, вызывающего функцию `get()`. Аналогично приведенный ниже оператор читает две следующих друг за другом строки в массивы `name1` и `name2`, что эквивалентно двум отдельным вызовам `cin.getline()`:

```
cin.getline(name1, ArSize).getline(name2, ArSize);
```

В листинге 4.5 применяется конкатенация. В главе 11 вы узнаете о том, как включить это средство в собственные определения классов.

Листинг 4.5. insrt3.cpp

```
// insrt3.cpp -- чтение более одного слова с помощью get() и get()
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";                 // запрос имени
    cin.get(name, ArSize).get();                  // читать строку и символ новой строки
    cout << "Enter your favorite dessert:\n";     // запрос любимого десерта
    cin.get(dessert, ArSize).get();
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Вот пример запуска программы из листинга 4.5:

```
Enter your name:
Mai Parfait
Enter your favorite dessert:
Chocolate Mousse
I have some delicious Chocolate Mousse for you, Mai Parfait.
```

Обратите внимание на то, что C++ допускает существование множества версий функции с разными списками аргументов.

Если вы используете, скажем, `cin.get(name, ArSize)`, то компилятор определяет, что вызывается форма, которая помещает строку в массив, и подставляет соответствующую функцию-член. Если же вместо этого вы применяете `cin.get()`, то компилятор видит, что вам нужна форма, которая читает один символ. В главе 8 это средство, называемое *перегрузкой функций*, рассматривается более подробно.

Зачем вообще может понадобиться вызывать `get()` вместо `getline()`? Во-первых, старые реализации могут не располагать `getline()`. Во-вторых, `get()` позволяет проявлять бóльшую осторожность. Предположим, например, что вы воспользовались `get()`, чтобы прочитать строку в массив. Как вы определите, была прочитана полная строка или же чтение прервалось в связи с заполнением массива? Для этого нужно посмотреть на следующий в очереди символ. Если это символ новой строки,

значит, была прочитана вся строка. В противном случае строка была прочитана не полностью и еще есть что читать. Этот прием исследуется в главе 17. Короче говоря, `getline()` немного проще в применении, но `get()` упрощает проверку ошибок. Вы можете использовать любую из этих функций для чтения ввода; просто учитывайте различия в их поведении.

Пустые строки и другие проблемы

Что происходит после того, как функции `getline()` и `get()` прочитали пустую строку? Изначально предполагалось, что следующий оператор ввода должен получить указание, где завершил работу предыдущий вызов `getline()` или `get()`. Однако современная практика заключается в том, что после того, как функция `get()` (но не `getline()`) прочитает пустую строку, она устанавливает флажок, который называется `failbit`. Влияние этого флажка состоит в том, что последующий ввод блокируется, но вы можете восстановить его следующей командой:

```
cin.clear();
```

Другая потенциальная проблема связана с тем, что входная строка может быть длиннее, чем выделенное для нее пространство. Если входная строка длиннее, чем указанное количество символов, то и `getline()`, и `get()` оставляют избыточные символы во входной очереди. Однако `getline()` дополнительно устанавливает `failbit` и отключает последующий ввод.

В главах 5, 6 и 17 эти свойства и способы программирования с их учетом рассматриваются более подробно.

Смешивание строкового и числового ввода

Смешивание числового и строкового ввода может приводить к проблемам. Рассмотрим пример простой программы в листинге 4.6.

Листинг 4.6. numstr.cpp

```
// numstr.cpp -- строковый ввод после числового
#include <iostream>
int main()
{
    using namespace std;
    cout << "What year was your house built?\n";   // ввод года постройки дома
    int year;
    cin >> year;
    cout << "What is its street address?\n";      // ввод адреса
    char address[80];
    cin.getline(address, 80);
    cout << "Year built: " << year << endl;        // вывод года постройки
    cout << "Address: " << address << endl;        // вывод адреса
    cout << "Done!\n";
    return 0;
}
```

В результате выполнения программы из листинга 4.6 получаем следующий вывод:

```
What year was your house built?
1966
What is its street address?
Year built: 1966
Address:
Done!
```

Вы так и не получили возможности ввести адрес. Проблема в том, что когда cin читает год, то оставляет символ новой строки, сгенерированный нажатием <Enter>, во входной очереди. Затем cin.getline() читает символ новой строки просто как пустую строку, после чего присваивает массиву address нулевую строку. Чтобы исправить это, нужно перед чтением адреса прочитать и отбросить символ новой строки. Это может быть сделано несколькими способами, включая вызов get() без аргументов либо с аргументом char, как описано в предыдущем примере. Эти вызовы можно выполнить по отдельности:

```
cin >> year;
cin.get(); // или cin.get(ch);
```

Или же можно сцепить вызов, воспользовавшись тем фактом, что выражение cin >> year возвращает объект cin:

```
(cin >> year).get(); // или (cin >> year).get(ch);
```

Если внести одно из таких исправлений в листинг 4.6, программа станет работать правильно:

```
What year was your house built?
1966
What is its street address?
43821 Unsigned Short Street
Year built: 1966
Address: 43821 Unsigned Short Street
Done!
```

Для обработки строк в программах на C++ часто используются указатели вместо массивов. Мы обратимся к этому аспекту строк после того, как немного поговорим об указателях. А пока рассмотрим более современный способ обработки строк: класс C++ по имени string.

Введение в класс string

В стандарте ISO/ANSI C++98 библиотека C++ была расширена за счет добавления класса string. Поэтому отныне вместо использования символьных массивов для хранения строк можно применять переменные типа string (или, пользуясь терминологией C++, объекты). Как вы увидите, класс string проще в использовании, чем массив, и к тому же предлагает более естественное представление строки как типа.

Для работы с классом string в программе должен быть включен заголовочный файл string. Класс string является частью пространства имен std, поэтому вы должны указать директиву using или объявление либо же ссылаться на класс как std::string. Определение класса скрывает природу строки как массива символов и позволяет трактовать ее как обычную переменную. В листинге 4.7 проиллюстрированы некоторые сходства и различия между объектами string и символьными массивами.

Листинг 4.7. strtype1.cpp

```
// strtype1.cpp -- использование класса C++ string
#include <iostream>
#include <string> // обеспечение доступа к классу string
int main()
{
   using namespace std;
```