11

Настройка ввода и вывода

“Все должно быть как можно более простым,
но не проще”.

Альберт Эйнштейн (Albert Einstein)

В этой главе мы обсудим, как адаптировать потоки ввода-вывода, описанные в главе 10, к конкретным потребностям и вкусам. Это связано со множеством деталей, которые обусловлены тем, как люди читают тексты, а также с ограничениями на использование файлов. Заключительный пример иллюстрирует проект потока ввода, в котором можно задавать собственный набор операторов.

В этой главе...

11.1. Регулярность и нерегулярность

Библиотека ввода-вывода является частью стандартной библиотеки языка C++. Она обеспечивает единообразную и расширяемую базу для ввода и вывода текста. Под словом “текст” мы подразумеваем нечто, что можно представить в виде последовательности символов. Таким образом, когда мы говорим о вводе и выводе, то целое число `1234` рассматривается как текст, поскольку его можно записать с помощью четырех символов: `1`, `2`, `3` и `4`.

До сих пор мы не делали различий между источниками входной информации. Однако иногда этого оказывается недостаточно. Например, файлы отличаются от других источников данных (например, линий связи), поскольку они допускают адресацию отдельных байтов. Кроме того, мы работали, основываясь на предположении, что тип объекта полностью определен схемой его ввода и вывода. Это не совсем правильно и совсем недостаточно. Например, при выводе мы часто хотим указывать количество цифр, используемых для представления числа с плавающей точкой (его точность). В данной главе описано много способов, с помощью которых можно настроить ввод и вывод для своих потребностей.

Будучи программистами, мы предпочитаем регулярность. Единообразная обработка всех объектов, находящихся в памяти, одинаковый подход ко всем источникам входной информации и стандартное унифицированное представление объектов при входе в систему и выходе из нее позволяют создавать самый ясный, простой, понятный и часто самый эффективный код. Однако наши программы должны служить людям, а люди имеют стойкие предпочтения. Таким образом, как программисты мы должны поддерживать баланс между сложностью программы и настройкой на персональные вкусы пользователей.

11.2. Форматирование вывода

Люди уделяют много внимания мелким деталям, связанным с представлением выходной информации, которую им необходимо прочитать. Например, для физика число `1.25` (округленное до двух цифр после точки) может сильно отличаться от числа `1.24670477`, а для бухгалтера запись `(1.25)` может сильно отличаться от записи `( 1.2467)` и совершенно не совпадать с числом `1.25` (в финансовых документах скобки иногда означают убытки, т.е. отрицательные величины). Как программисты мы стремимся сделать наш вывод как можно более ясным и как можно более близким к ожиданиям потребителей нашей программы. Потоки вывода (`ostream`) предоставляют массу возможностей для форматирования вывода данных, имеющих встроенные типы. Для типов, определенных пользователем, программист сам должен определить подходящие операции `<<`.

Количество деталей, уточнений и возможностей при выводе кажется неограниченным, а при вводе, наоборот, есть лишь несколько вариантов. Например, для обозначения десятичной точки можно использовать разные символы (как правило, точку или запятую), денежные суммы в разных валютах также выводятся по-разному, а истинное логическое значение можно выражать как словом `true` (или `vrai` or `sandt`), так и числом `1`, если в вашем распоряжении находятся только символы, не входящие в набор ASCII (например, символы в системе Unicode). Кроме того, существуют разные способы ограничения символов, записываемых в строку. Эти возможности не интересны, пока они вам не нужны, поэтому мы отсылаем читателей к справочникам и специализированным книгам, таким как Langer *Standard C++ IOStreams and Locales*; главе 21 и приложению D в книге *The C++ Programming Language* Страуструпа; а также к §22 и 27 стандарта ISO C++. В настоящей книге мы рассмотрим лишь самые распространенные варианты вывода и некоторые общие понятия.

11.2.1. Вывод целых чисел

Целые числа можно вывести как восьмеричные (в системе счисления с основанием 8), десятичные (в обычной системе счисления с основанием 10) и шестнадцатеричные (в системе счисления с основанием 16). Если вы ничего не знаете об этих системах, сначала прочитайте раздел A.1.2.1. В большинстве случаев при выводе используется десятичная система. Шестнадцатеричная система широко распространена при выводе информации, связанной с аппаратным обеспечением.

Причина популярности шестнадцатеричной системы кроется в том, что шестнадцатеричные цифры позволяют точно представить четырехбитовые значения. Таким образом, две шестнадцатеричные цифры можно использовать для представления восьмибитового байта, четыре шестнадцатеричные цифры представляют два байта (которые часто являются полусловом), восемь шестнадцатеричных цифр могут представить четыре байта (что часто соответствует размеру слова или регистра).

Когда был разработан язык C — предшественник языка C++ (в 1970-х годах), не менее популярной была восьмеричная система, но сейчас она используется редко.

Мы можем указать, что (десятичное число) **1234** при выводе должно трактоваться как десятичное, шестнадцатеричное или восьмеричное.

```
cout << 1234 << "\t(decimal)\n"
     << hex << 1234 << "\t(hexadecimal)\n"
     << oct << 1234 << "\t(octal)\n";
```

Символ **'\t'** означает "символ табуляции". Он обеспечивает следующее представление выходной информации:

```
1234    (decimal)
4d2     (hexadecimal)
2322    (octal)
```

Обозначения **<< hex** и **<< oct** не являются значениями, предназначенными для вывода. Выражение **<< hex** сообщает потоку, что любое целое число в дальнейшем должно быть представлено как шестнадцатеричное, а выражение **<< oct** означает, что любое целое число в дальнейшем должно быть представлено как восьмеричное. Рассмотрим пример.

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << 1234 << '\n'; // восьмеричная основа продолжает действовать
```

В итоге получаем следующий вывод:

```
1234 4d2 2322
2322 // целые числа продолжают трактоваться как восьмеричные
```

Обратите внимание на то, что последнее число выведено как восьмеричное; иначе говоря, термины **oct**, **hex** и **dec** (для десятичных чисел) являются персистентными (инертными) — они применяются к каждому целому числу, пока мы не дадим потоку другое указание. Термины **hex** и **oct** используются для изменения поведения потока и называются *манипуляторами* (manipulators).

↘ ПОПРОБУЙТЕ

Выведите ваш день рождения в десятичном, восьмеричном и шестнадцатеричном форматах. Обозначьте каждое из этих значений. Выровняйте ваш вывод по столбцам, используя символ табуляции, и выведите свой возраст.

Представление чисел в системе счисления, отличной от десятичной, может ввести читателя в заблуждение. Например, если заранее не знать, в какой системе представлено число, то строка **11** может означать десятичное число 11, а не восьмеричное число 9 (т.е. **11** в восьмеричной системе) или шестнадцатеричное число 17 (т.е. **11** в шестнадцатеричной системе). Для того чтобы избежать таких проблем, можно попросить поток показать базу, в которой представлено целое число. Рассмотрим пример.

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << showbase << dec; // показывать базы
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
```

В результате получим следующий вывод:

```
1234 4d2 2322
1234 0x4d2 02322
```

Итак, десятичные числа не имеют префиксов, восьмеричные числа имеют префикс `0`, а шестнадцатеричные числа имеют префикс **`0x`** (или **`0X`**). Именно так обозначаются целочисленные литералы в языке C++. Рассмотрим пример.

```
cout << 1234 << '\t' << 0x4d2 << '\t' << 02322 << '\n';
```

В десятичном виде эти числа выглядели бы так:

```
1234 1234 1234
```

Как вы могли заметить, манипулятор **`showbase`** является персистентным, как и манипуляторы **`oct`** и **`hex`**. Манипулятор **`noshowbase`** отменяет действие манипулятора **`showbase`**, возвращая поток в состояние по умолчанию, в котором любое число выводится без указания его базы счисления.

Итак, существует несколько манипуляторов вывода.

Манипуляторы для вывода целых чисел	
`oct`	Использовать восьмеричную систему счисления
`dec`	Использовать десятичную систему счисления
`hex`	Использовать шестнадцатеричную систему счисления
`showbase`	Префикс `0` для восьмеричных и `0x` для шестнадцатеричных
`noshowbase`	Не использовать префиксы

11.2.2. Ввод целых чисел

По умолчанию оператор **`>>`** предполагает, что числа используются в десятичной системе счисления, но его можно заставить вводить целые числа как шестнадцатеричные или восьмеричные.

```
int a;
int b;
int c;
int d;
cin >> a >> hex >> b >> oct >> c >> d;
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

Если набрать на клавиатуре числа

```
1234 4d2 2322 2322
```

то программа выведет их так:

```
1234 1234 1234 1234
```

Обратите внимание на то, что при вводе манипуляторы **oct**, **dec** и **hex** являются персистентными, как и при выводе.

➘ ПОПРОБУЙТЕ

Завершите фрагмент кода, приведенный выше, и преобразуйте его в программу. Попробуйте ввести предлагаемые числа; затем введите числа

```
1234 1234 1234 1234
```

Объясните результат. Попробуйте ввести другие числа, чтобы увидеть, что произойдет.

Для того чтобы принять и правильно интерпретировать префиксы **0** и **0x**, можно использовать оператор **>>**. Для этого необходимо отменить установки, принятые по умолчанию. Рассмотрим пример.

```
cin.unsetf(ios::dec); // не считать десятичным
                      // (т.е. 0x может означать
                      // шестнадцатеричное число)
cin.unsetf(ios::oct); // не считать восьмеричным
                      // (т.е. 12 может означать двенадцать)
cin.unsetf(ios::hex); // не считать шестнадцатеричным
                      // (т.е. 12 может означать двенадцать)
```

Функция-член потока **unsetf()** сбрасывает флаг (или флаги), указанный как аргумент. Итак, если вы напишете

```
cin >>a >> b >> c >> d;
```

и введете

```
1234 0x4d2 02322 02322
```

то получите

```
1234 1234 1234 1234
```

11.2.3. Вывод чисел с плавающей точкой

Если вы непосредственно работаете с аппаратным обеспечением, то вам нужны шестнадцатеричные числа (и, возможно, восьмеричные). Аналогично, если вы проводите научные вычисления, то должны форматировать числа с плавающей точкой. Они обрабатываются манипуляторами потока **iostream** почти так, как и десятичные числа. Рассмотрим пример.

```
cout << 1234.56789 << "\t\t(общий)\n" // \t\t — выравнивание столбцов
     << fixed << 1234.56789 << "\t(фиксированный)\n"
     << scientific << 1234.56789 << "\t(научный)\n";
```

В итоге получим следующие строки:

```
1234.57          (общий)
1234.567890      (фиксированный)
1.234568e+003    (научный)
```

Манипуляторы **fixed** и **scientific** используются для выбора форматов для представления чисел с плавающей точкой. Интересно, что в стандартной библиотеке нет манипулятора **general**, который устанавливал бы формат, принятый по умолчанию. Однако мы можем определить его сами, как это сделано в заголовочном файле **std_lib_facilities.h**. Для этого не требуются знания о внутреннем устройстве библиотеки ввода-вывода.

```
inline ios_base& general(ios_base& b) // фиксированный и научный
                                      // формат
     // сбрасывает все флаги формата с плавающей точкой
{
     b.setf(ios_base::fmtflags(0), ios_base::floatfield);
     return b;
}
```

Теперь можем написать следующий код:

```
cout << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << 1234.56789 << '\n';           // действует формат
                                      // с плавающей точкой
cout << general << 1234.56789 << '\t' // предупреждение:
     << fixed << 1234.56789 << '\t'   // general — нестандартный
                                      // манипулятор
     << scientific << 1234.56789 << '\n';
```

В итоге получим следующие числа:

```
1234.57 1234.567890 1.234568e+003
1.234568e+003 // манипулятор научного формата является
              // персистентным
1234.57 1234.567890 1.234568e+003
```

Итак, существует несколько манипуляторов для работы с числами с плавающей точкой.

Формат чисел с плавающей точкой	
`fixed`	Использовать представление с фиксированной точкой
`scientific`	Использовать мантиссу и показатель степени; мантисса всегда лежит в диапазоне [1:10), т.е. перед десятичной точкой всегда стоит ненулевая десятичная цифра
`general`	Выбирает манипулятор `fixed` или `scientific` для наиболее точного представления чисел в рамках точности самого манипулятора `general`. Формат `general` принят по умолчанию, но для его явного определения `general()`

11.2.4. Точность

По умолчанию число с плавающей точкой выводится на печать с помощью шести цифр в формате **general**. Формат, состоящий из шести цифр (точность формата

general по умолчанию), считается наиболее подходящим, а такое округление числа — наилучшим. Рассмотрим пример.

1234.567 выводится на печать как **1234.57**

1.2345678 выводится на печать как **1.23457**

Округление, как правило, выполняется по правилу 4/5: от 0 до 4 — округление вниз, а от 5 до 9 — вверх. Обратите внимание на то, что такое форматирование относится только к числам с плавающей точкой.

1234567 выводится на печать как **1234567** (поскольку число целое)

1234567.0 выводится на печать как **1.23457e+006**

В последнем случае поток **ostream** распознает, что число **1234567.0** нельзя вывести на печать в формате **fixed**, используя только шесть цифр, и переключается на формат **scientific**, чтобы обеспечить как можно более точное представление числа. В принципе формат **general** может автоматически заменяться форматами **scientific** и **fixed**, чтобы обеспечить максимально точное представление числа с плавающей точкой в рамках общего формата, предусматривающего использование шести цифр.

➘ ПОПРОБУЙТЕ

Напишите программу, три раза выводящую на печать число **1234567.89**, сначала в формате **general**, затем — в **fixed**, потом — в **scientific**. Какая форма вывода обеспечивает наиболее точное представление числа и почему?

Программист может установить точность представления числа, используя манипулятор **setprecision()**. Рассмотрим пример.

```
cout << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << general << setprecision(5)
     << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << general << setprecision(8)
     << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
```

Этот код выводит на печать следующие числа (обратите внимание на округление):

```
1234.57      1234.567890   1.234568e+003
1234.6 1234.56789    1.23457e+003
1234.5679     1234.56789000       1.23456789e+003
```

Точность определятся по правилам, приведенным ниже.

Точность чисел с плавающей точкой	
`general`	Точность определяется общим количеством цифр
`scientific`	Точность определяется количеством цифр после десятичной точки
`fixed`	Точность определяется количеством цифр после десятичной точки

Мы рекомендуем использовать формат, принятый по умолчанию (формат **`general`** с точностью, равной шести цифрам), если у вас нет весомых причин для применения другого формата. Обычно причина, по которой выбираются другие форматы, такова: “Мы хотим получить б*о*льшую точность при выводе”.

11.2.5. Поля

С помощью научного и фиксированного формата программист может точно контролировать, сколько места займет число на выходе. Это очень полезно при распечатке таблиц и т.п. Эквивалентный механизм для целых чисел называют *полями* (fields). Вы можете точно указать ширину поля, используя манипулятор **`setw()`**. Рассмотрим пример.

```
cout << 123456                              // поля не используются
     <<'|'<< setw(4) << 123456 << '|'       // число 123456
                                            // не помещается в поле
     << setw(8) << 123456 << '|'            // из 4 символов,
                                            // расширим до 8
     << 123456 << "|\n";                    // размеры полей не инертны
```

В итоге получим следующий результат:

```
123456|123456|  123456|123456|
```

Обратите внимание на два пробела перед третьим появлением числа **`123456`**. Это является результатом того, что мы выводим шесть цифр в поле, состоящее из восьми символов. Однако число **`123456`** невозможно усечь так, чтобы оно помещалось в поле, состоящее из четырех символов. Почему? Конечно, числа **`|1234|`** или **`|3456|`** можно интерпретировать как вполне допустимые для поля, состоящего из четырех символов. Однако в этом случае на печать будут выведены числа, которые совершенно не соответствуют ожиданиям программиста, причем он не получит об этом никакого предупреждения. Поток **`ostream`** не сделает этого; вместо этого он аннулирует неправильный формат вывода. Плохое форматирование почти всегда лучше, чем “плохие результаты”. В большинстве случаев (например, при выводе таблиц) переполнение полей сразу бросается в глаза и может быть исправлено.

Поля также можно использовать при выводе строк и чисел с плавающей точкой. Рассмотрим пример.

```
cout << 12345 <<'|'<< setw(4) << 12345 << '|'
     << setw(8) << 12345 << '|' << 12345 << "|\n";
cout << 1234.5 <<'|'<< setw(4) << 1234.5 << '|'
```

```
     << setw(8) << 1234.5 << '|' << 1234.5 << "|\n";
cout << "asdfg" <<'|'<< setw(4) << "asdfg" << '|'
     << setw(8) << "asdfg" << '|' << "asdfg" << "|\n";
```

Этот код выводит на печать следующие числа:

```
12345|12345| 12345|12345|
1234.5|1234.5| 1234.5|1234.5|
asdfg|asdfg| asdfg|asdfg|
```

Обратите внимание на то, что ширина поля не является инертным параметром. Во всех трех случаях первое и последнее числа по умолчанию выведены с максимальным количеством цифр, которые допускает текущий формат. Иначе говоря, если мы непосредственно перед выводом не укажем ширину поля, то понятие поля вообще не будет использовано.

➘ ПОПРОБУЙТЕ

Создайте простую таблицу, содержащую фамилию, имя, номер телефона и адрес электронной почты не менее пяти ваших друзей. Поэкспериментируйте с разной шириной поля, пока не найдете приемлемый вид таблицы.

11.3. Открытие файла и позиционирование

В языке C++ файл — это абстракция возможностей операционной системы. Как указано в разделе 10.3, файл — это последовательность байтов, пронумерованных начиная с нуля.

Вопрос заключается лишь в том, как получить доступ к этим байтам. При работе с потоками **iostream** вид доступа определяется в тот момент, когда мы открываем файл и связываем с ним поток. Поток сам определяет, какие операции можно выполнить после открытия файла и каков их смысл. Например, когда мы открываем для файла поток **istream**, то можем прочитать его содержимое, а когда открываем для файла поток **ostream**, то можем записать в него данные.

11.3.1. Режимы открытия файлов

Файл можно открыть в одном из нескольких режимов. По умолчанию поток **ifstream** открывает файлы для чтения, а поток **ofstream** — для записи. Эти операции удовлетворяют большинство наших потребностей. Однако существует несколько альтернатив.

Режимы открытия файлов	
`ios_base::app`	Добавить (т.е. приписать в конце файла)
`ios_base::ate`	В конец (открыть и перейти в конец файла)
`ios_base::binary`	Бинарный режим — зависит от специфики системы
`ios_base::in`	Для чтения
`ios_base::out`	Для записи
`ios_base::trunc`	Обрезать файл до нулевой длины

Режим открытия файла можно указать после его имени. Рассмотрим пример.

```
ofstream of1(name1); // по умолчанию ios_base::out
ifstream if1(name2); // по умолчанию ios_base::in
ofstream ofs(name, ios_base::app); // по умолчанию ofstream —
                                   // для записи
fstream fs("myfile", ios_base::in|ios_base::out); // для ввода и вывода
```

Символ | в последнем примере — это побитовый оператор ИЛИ (раздел A.5.5), который можно использовать для объединения режимов. Опция **app** часто используется для записи регистрационных файлов, в которых записи всегда добавляются в конец.

В любом случае конкретный режим открытия файла может зависеть от операционной системы. Если операционная система не может открыть файл в требуемом режиме, то поток перейдет в неправильное состояние.

```
if (!fs) // ой: мы не можем открыть файл в таком режиме
```

В большинстве ситуаций причиной сбоя при открытии файла для чтения является его отсутствие.

```
ifstream ifs("redungs");
if (!ifs) // ошибка: невозможно открыть файл readings для чтения
```

В данном случае причиной ошибки стала опечатка.

Обычно, когда вы пытаетесь открыть несуществующий файл, операционная система создает новый файл для вывода, но, к счастью, она не делает этого, когда вы обращаетесь к несуществующему файлу для ввода.

```
ofstream ofs("no-such-file");    // создает новый файл no-such-file
ofstream ifs("no-file-of-this-name"); // ошибка: поток ifs не нахо-
                                      // дится в состоянии good()
```

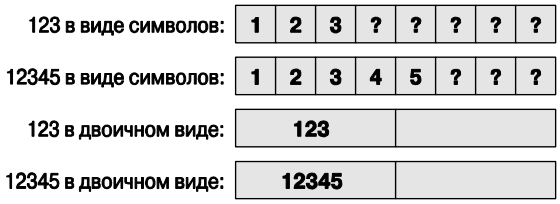
11.3.2. Бинарные файлы

В памяти мы можем представить значение 123 как целое или как строку. Рассмотрим пример.

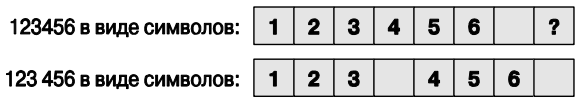
```
int n = 123;
string s = "123";
```

В первом случае число **123** интерпретируется как (двоичное) число. Объем памяти, который оно занимает, совпадает с объемом памяти, который занимает любое

другое целое число (4 байта, т.е. 32 бита на персональном компьютере). Если вместо числа **123** мы выберем число **12345**, то оно по-прежнему будет занимать те же самые четыре байта. Во втором варианте значение **123** хранится как строка из трех символов. Если мы выберем строку **"12345"**, то для ее хранения нам потребуются пять символов (плюс накладные расходы памяти на управление объектом класса **string**). Проиллюстрируем сказанное, используя обычные десятичное и символьное представления, а не двоичное, как в памяти компьютера.



Когда мы используем символьное представление, то какой-то символ должен служить признаком конца числа, так же как на бумаге, когда мы записываем одно число 123456 и два числа 123 456. На бумаге для разделения чисел мы используем пробел. То же самое можно сделать в памяти компьютера.



Разница между хранением двоичного представления фиксированного размера (например, в виде типа **int**) и символьного представления переменного размера (например, в виде типа **string**) проявляется и при работе с файлами. По умолчанию потоки **iostream** работают с символьными представлениями; иначе говоря, поток **istream** считывает п — оследовательность символов и превращает их в объект заданного типа. Поток **ostream** принимает объект заданного типа и преобразует их в последовательность записываемых символов. Однако можно потребовать, чтобы потоки **istream** и **ostream** просто копировали байты из файла в файл. Такой ввод-вывод называется *двоичным* (binary I/O). В этом случае файл необходимо открыть в режиме **ios_base::binary**. Рассмотрим пример, в котором считываются и записываются двоичные файлы, содержащие целые числа. Главные сроки, предназначенные для обработки двоичных файлов, объясняются ниже.

```
int main()
{
     // открываем поток istream для двоичного ввода из файла:
     cout << "Пожалуйста. введите имя файла для ввода\n";
     string name;
     cin >> name;
     ifstream ifs(name.c_str(),ios_base::binary); // примечание: опция
          // binary сообщает потоку, чтобы он ничего не делал
          // с байтами
```

```
    if (!ifs) error("невозможно открыть файл для ввода ", name);

    // открываем поток ostream для двоичного вывода в файл:
    cout << "Пожалуйста, введите имя файла для вывода\n";
    cin >> name;
    ofstream ofs(name.c_str(),ios_base::binary); // примечание: опция
        // binary сообщает потоку, чтобы он ничего не делал
        // с байтами
    if (!ofs) error("невозможно открыть файл для ввода ",name);

    vector<int> v;
    // чтение из бинарного файла:
    int i;
    while (ifs.read(as_bytes(i),sizeof(int))) // примечание:
                                              // читаем байты
    v.push_back(i);
    // . . . что-то делаем с вектором v . . .

    // записываем в двоичный файл:
    for(int i=0; i<v.size(); ++i)
        ofs.write(as_bytes(v[i]),sizeof(int)); // примечание:
                                               // запись байтов
    return 0;
}
```

Мы открыли эти файлы с помощью опции **ios_base::binary** .

```
ifstream ifs(name.c_str(), ios_base::binary);
ofstream ofs(name.c_str(), ios_base::binary);
```

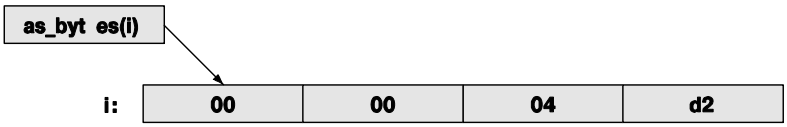
В обоих вариантах мы выбрали более сложное, но часто более компактное двоичное представление. Если мы перейдем от символьно-ориентированного ввода-вывода к двоичному, то не сможем использовать обычные операторы ввода и вывода **>>** и **<<**. Эти операторы преобразуют значения в последовательности символов, руководствуясь установленными по умолчанию правилами (например, строка **"asdf"** превращается в символы **a**, **s**, **d**, **f**, а число **123** превращается в символы **1**, **2**, **3**). Если вы не хотите работать с двоичным представлением чисел, достаточно ничего не делать и использовать режим, заданный по умолчанию. Мы рекомендуем применять опцию **binary**, только если вы (или кто-нибудь еще) считаете, что так будет лучше. Например, с помощью опции **binary** можно сообщить потоку, что он ничего не должен делать с байтами.

А что вообще мы могли бы сделать с типом **int**? Очевидно, записать его в память размером четыре байта; иначе говоря, мы могли бы обратиться к представлению типа **int** в памяти (последовательность четырех байтов) и записать эти байты в файл. Позднее мы могли бы преобразовать эти байты обратно в целое число.

```
ifs.read(as_bytes(i),sizeof(int))      // чтение байтов
ofs.write(as_bytes(v[i]),sizeof(int)) // запись байтов
```

Функция **write()** потока **ostream** и функция **read()** потока **istream** принимают адрес (с помощью функции **as_byte()**) и количество байтов (символов), полученное

с помощью оператора **sizeof**. Этот адрес должен ссылаться на первый байт в памяти, хранящей значение, которое мы хотим прочитать или записать. Например, если у нас есть объект типа **int** со значением **1234**, то мы могли бы получить четыре байта (используя шестнадцатеричную систему обозначений) — **00**, **00**, **04**, **d2**:



Функция **as_bytes()** позволяет получить адрес первого байта объекта. Ее определение выглядит так (некоторые особенности языка, использованные здесь, будут рассмотрены в разделах 17.8 и 19.3):

```
template<class T>
char* as_bytes(T& i) // рассматривает объект T как последовательность
                     // байтов
{
     void* addr = &i; //получаем адрес первого байта
                      //памяти, использованной для хранения объекта
     return static_cast<char*>(addr); // трактуем эту память как байты
}
```

Небезопасное преобразование типа с помощью оператора **static_cast** необходимо для того, чтобы получить переменную в виде совокупности байтов. Понятие адреса будет подробно изучено в главах 17 и 18. Здесь мы просто показываем, как представить любой объект, хранящийся в памяти, в виде совокупности байтов, чтобы прочитать или записать его с помощью функций **read()** и **write()**.

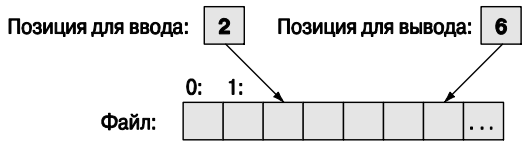
Этот двоичный вывод запутан, сложен и уязвим для ошибок. Однако программисты не всегда должны иметь полную свободу выбора формата файла, поэтому иногда они просто вынуждены использовать двоичный ввод-вывод по воле кого-то другого. Кроме того, отказ от символьного представления иногда можно логично обосновать. Типичные примеры — рисунок или звуковой файл, — не имеющие разумного символьного представления: фотография или фрагмент музыкального произведения по своей природе является совокупностью битов.

Символьный ввод-вывод, по умолчанию предусмотренный в библиотеке, не изменяется при переносе программ из одного компьютера в другой, доступен для человеческого понимания и поддерживается любыми средствами набора текстов. Если есть возможность, рекомендуем использовать именно символьный ввод-вывод, а двоичный ввод-вывод применять только в случае крайней необходимости.

11.3.3. Позиционирование в файлах

При малейшей возможности считывайте и записывайте файлы от начала до конца. Это проще всего и открывает меньше возможностей для совершения

ошибок. Каждый раз, когда вы понимаете, что пора изменить файл, лучше создайте новый и запишите в него все изменения. Однако, если вы должны поступить иначе, то можно выполнить позиционирование и указать конкретное место для чтения и записи в файле. В принципе в любом файле, открытом для чтения, существует позиция для считывания/ввода (“read/get position”), а в любом файле, открытом для записи, есть позиция для записи/вывода (“write/put position”).



Эти позиции можно использовать следующим образом.

```
fstream fs(name.c_str()); // открыть для ввода и вывода
if (!fs) error("can't open ",name);

fs.seekg(5); // перенести позицию считывания (буква g означает "get")
             // на пять ячеек вперед (шестой символ)
char ch;
fs>>ch;      // считать и увеличить номер позиции для считывания
cout << "шестой символ – это " << ch << '(' << int(ch) << ")\n";
fs.seekp(1); // перенести позицию для записи (буква p означает "put")
             // на одну ячейку вперед
fs<<'y';     // записать и увеличить позицию для записи
```

Будьте осторожны: ошибки позиционирования не распознаются. В частности, если вы попытаетесь выйти за пределы файла (используя функцию **seekg()** или **seekp()**), то последствия могут быть непредсказуемыми и состояние операционной системы изменится.

11.4. Потоки строк

☑ В качестве источника ввода для потока **istream** или цели вывода для потока **ostream** можно использовать объект класса **string**. Поток **istream**, считывающий данные из объекта класса **string**, называется **istringstream**, а поток **ostream**, записывающий символы в объект класса **string**, называется **ostringstream**. Например, поток **istringstream** полезен для извлечения числовых значений из строк.

```
double str_to_double(string s)
     // если это возможно, преобразовывает символы из строки s
     // в число с плавающей точкой
{
     istringstream is(s); // создаем поток для ввода из строки s
     double d;
     is >> d;
     if (!is) error("ошибка форматирования типа double: ",s);
```

```
    return d;
}
    double d1 = str_to_double("12.4"); // проверка
    double d2 = str_to_double("1.34e-3");
    double d3 = str_to_double("twelve point three"); // вызывается
                                                     // error()
```

Если попытаться прочитать данные за пределами строки, предназначенной для ввода в поток **stringstream**, то он перейдет в состояние **eof()**. Это значит, что для потока **stringstream** можно использовать обычный цикл ввода; строковый поток на самом деле является разновидностью потока **istream**.

Поток **ostringstream**, наоборот, может быть полезен для форматирования вывода в системах, ожидающих аргумента в виде простой строки, например в системах графического пользовательского интерфейса (раздел 16.5). Рассмотрим пример.

```
void my_code(string label, Temperature temp)
{
    // . . .
    ostringstream os; // поток для составления сообщения
    os << setw(8) << label << ": "
       << fixed << setprecision(5) << temp.temp << temp.unit;
    someobject.display(Point(100,100), os.str().c_str());
    // . . .
}
```

Функция-член **str()** класса **ostringstream** возвращает объект класса **string**, составленный операторами вывода, в поток **ostringstream**. Функция **c_str()** — это функция-член класса **string**, возвращающая строки в стиле языка C, которые ожидаются интерфейсами многих систем.

Потоки **stringstream** обычно используются, когда мы хотим отделить собственно ввод-вывод от обработки данных. Например, аргумент типа **string** в функции **str_to_double()** обычно поступает из файла (например, из журнала событий веб) или с клавиатуры. Аналогично, сообщение, составленное функцией **my_code()**, в конце концов выводится на экран. Например, в разделе 11.7 мы используем поток **stringstream** при выводе для фильтрации нежелательных символов. Таким образом, потоки **stringstream** можно интерпретировать как механизм настройки ввода-вывода для особых потребностей и вкусов.

Продемонстрируем использование потока **ostream** на простом примере конкатенации строк.

```
int seq_no = get_next_number(); // вводим число из системного журнала
ostringstream name;
name << "myfile" << seq_no; // например, myfile17
ofstream logfile(name.str().c_str()); // например, открыть myfile17
```

Как правило, поток **istringstream** инициализируется объектом класса **string**, а затем считывает из него символы, используя операторы ввода. И наоборот, поток **ostringstream** инициализируется пустым объектом класса **string**, а затем запол-

няется с помощью операторов вывода. Существует более простой способ доступа к символам в потоке **stringstream**, который иногда оказывается полезным: функция **ss.str()** возвращает копию строки из объекта **ss**, а функция **ss.str(s)** присваивает строке в объекте **ss** копию строки **s**. В разделе 11.7 приведен пример, в котором функция **ss.str(s)** играет существенную роль.

11.5. Ввод, ориентированный на строки

Оператор **>>** вводит данные в объекты заданного типа в соответствии со стандартным форматом, установленным для этого типа. Например, при вводе чисел в объект типа **int** оператор **>>** будет выполнять ввод, пока не обнаружит символ, не являющийся цифрой, а при вводе в объект класса **string** оператор **>>** будет считывать символы, пока не обнаружит разделитель (whitespace). Стандартная библиотека **istream** содержит также средства для ввода отдельных символов и целых строк. Рассмотрим пример.

```
string name;
cin >> name;            // ввод: Dennis Ritchie
cout << name << '\n';   // вывод: Dennis
```

Что, если мы захотим прочитать всю строку сразу, а способ ее форматирования выберем потом? Это можно сделать с помощью функции **getline()**. Рассмотрим пример.

```
string name;
getline(cin,name);      // ввод: Dennis Ritchie
cout << name << '\n';   // вывод: Dennis Ritchie
```

Теперь мы считали целую строку. Зачем нам это было нужно? Например, неплохой ответ: “Потому что мы сделали то, чего не может оператор **>>**”. Часто можно слышать совершенно неудачное объяснение: “Потому что пользователь набрал полную строку”. Если это все, что вы можете сказать, то используйте оператор **>>**, потому что, если вы ввели строку, то должны как-то ее разобрать на части. Рассмотрим пример.

```
string first_name;
string second_name;
stringstream ss(name);
ss>>first_name;        // ввод строки Dennis
ss>>second_name;       // ввод строки Ritchie
```

Непосредственный ввод данных в строки **first_name** и **second_name** можно было бы выполнить проще. Одна из распространенных причин для считывания полной строки заключается в том, что определение разделителя не всегда является достаточно приемлемым. Иногда переход на новую строку желательно трактовать не как разделитель. Например, в ходе обмена сообщениями в компьютерной игре

текст разумнее интерпретировать как предложение, не полагаясь на общепринятую пунктуацию.

```
идти налево, пока не увидишь картину справа на стене
сними картину со стены и открой дверь позади нее. Возьми сундук
```

В данном случае мы сначала прочитаем всю строку, а затем извлечем из нее отдельные слова.

```
string command;
getline(cin,command);                // вводим строку

stringstream ss(command);
vector<string> words;
string s;
while (ss>>s) words.push_back(s);  // извлекаем отдельные слова
```

С другой стороны, если есть выбор, то лучше всего ориентироваться на знаки пунктуации, а не на символ перехода на новую строку.

11.6. Классификация символов

Как правило, мы вводим целые числа, числа с плавающей точкой, слова и так далее, в соответствии с общепринятым форматом. Однако мы можем, а иногда и должны, снизить уровень абстракции и ввести отдельные символы. Для этого необходимо затратить больше усилий, но, считывая отдельные символы, мы получаем полный контроль на тем, что делаем. Рассмотрим задачу распознавания лексем в выражениях из раздела 7.8.2.

Допустим, мы хотим разделить выражение `1+4*x<=y/z*5` на одиннадцать лексем.

```
1 + 4 * x <= y / z * 5
```

Для ввода чисел мы могли бы использовать оператор `>>`, но, пытаясь ввести идентификаторы как строки, должны были бы прочитать фразу `x<=y` как целую строку (поскольку символы `<` и `=` не являются разделителями). Сочетание символов `z*` мы также должны были бы ввести как целую строку (поскольку символ `*` также не является разделителем).

Вместо этого можно сделать следующее:

```
char ch;
while (cin.get(ch)) {
    if (isspace(ch)) { // если символ ch является разделителем,
                       // ничего не делаем (так как разделители
                       // игнорируются)
    }
    if (isdigit(ch)) {
          // вводим число
    }
    else if (isalpha(ch)) {
          // вводим идентификатор
```

```
    }
    else {
          // обрабатываем операторы
    }
}
```

Функция **istream::get()** считывает отдельный символ в свой аргумент. Разделители при этом не игнорируются. Как и оператор **>>**, функция **get()** возвращает ссылку на свой поток **istream**, так что можно проверить его состояние.

При вводе отдельных символов мы обычно хотим классифицировать их: это символ или цифра? В верхнем регистре или в нижнем? И так далее. Для этого существует набор стандартных библиотечных функций.

Классификация символов

`isspace(c)`	Является ли `c` разделителем (`' '`, `'\t'`, `'\n'` и т.д.)?
`isalpha(c)`	Является ли `c` буквой (`'a'..'z'`, `'A'..'Z'`) (примечание: не `'_'`)?
`isdigit(c)`	Является ли `c` десятичной цифрой (`'0'..'9'`)?
`isxdigit(c)`	Является ли `c` шестнадцатеричной цифрой (десятичной цифрой или символом `'a'..'f'` или `'A'..'F'`)?
`isupper(c)`	Является ли `c` буквой в верхнем регистре?
`islower(c)`	Является ли `c` буквой в нижнем регистре?
`isalnum(c)`	Является ли `c` буквой или десятичной цифрой?
`iscntrl(c)`	Является ли `c` управляющим символом (ASCII 0..31 и 127)?
`ispunct(c)`	Правда ли, что `c` не является ни буквой, ни цифрой, ни разделителем, ни невидимым управляющим символом?
`isprint(c)`	Выводится ли символ `c` на печать (ASCII `' '..'~'`)?
`isgraph(c)`	Выполняется ли для `c` условие `isalpha()|isdigit()|ispunct()` (примечание: не пробел)?

Обратите внимание на то, что категории классификации можно объединять с помощью оператора ИЛИ (|). Например, выражение **isalnum(c)** означает **isalpha(c)**|**isdigit(c)**; иначе говоря, "является ли символ **c** буквой или цифрой?"

Кроме того, в стандартной библиотеке есть две полезные функции для уничтожения различий между символами, набранными в разных регистрах.

Регистр символа

`toupper(c)`	`c` или его эквивалент в верхнем регистре
`tolower(c)`	`c` или его эквивалент в нижнем регистре

Это удобно, когда мы хотим устранить различия между символами, набранными в разных регистрах. Например, если пользователь ввел слова **Right**, **right** и **rigHT**, то, скорее всего, он имел в виду одно и то же (например, слово **rigHT** чаще всего является результатом нечаянного нажатия клавиши <Caps Lock>). Применив функцию **tolower()** к каждому символу в каждой из строк, мы можем получить одно

и то же значение: **right**. Эту операцию можно выполнить с любым объектом класса **string**.

```
void tolower(string& s) // вводит строку s в нижнем регистре
{
     for (int i=0; i<s.length(); ++i) s[i] = tolower(s[i]);
}
```

Для того чтобы действительно изменить объект класса **string**, используем передачу аргумента по ссылке (см. раздел 8.5.5). Если бы мы хотели сохранить старую строку без изменения, то могли бы написать функцию, создающую ее копию в нижнем регистре. Мы предпочитаем функцию **tolower()**, а не **toupper()**, поскольку она лучше работает с текстами на некоторых естественных языках, например немецком, в которых не каждый символ в нижнем регистре имеет эквивалент в верхнем регистре.

11.7. Использование нестандартных разделителей

В этом разделе мы рассмотрим гипотетические примеры использования потоков **iostream** для решения реальных задач. При вводе строк слова по умолчанию разделяются пробелами или другими специальными символами (whitespace). К сожалению, поток **istream** не имеет средств, позволяющих определять, какие символы должны играть роль разделителей, или непосредственно изменять способ, с помощью которого оператор **>>** считывает строки. Итак, что делать, если мы хотим дать другое определение разделителю? Рассмотрим пример из раздела 4.6.3, в котором мы считывали слова и сравнивали их друг с другом. Между этими словами стояли разделители, поэтому если мы вводили строку

```
As planned, the guests arrived; then
```

то получали слова

```
As
planned,
the
guests
arrived;
then,
```

Это слова невозможно найти в словаре: “planned,” и “arrived;” — это вообще не слова. Это набор букв, состоящий из слов, к которым присоединены лишние и не относящиеся к делу знаки пунктуации. В большинстве случаев мы должны рассматривать знаки пунктуации как разделители. Как же избавиться от этих знаков пунктуации? Мы могли бы считать символы, удалить знаки пунктуации или преобразовать их в пробелы, а затем ввести “очищенные” данные снова.

```
string line;
getline(cin,line);                  // вводим строку line
```

```
for (int i=0; i<line.size(); ++i) //заменяем знаки пунктуации
                                  //пробелами

     switch(line[i]) {
     case ';': case '.': case ',': case '?': case '!':
          line[i] = ' ';
     }

stringstream ss(line); // создаем поток istream ss, вводя в него
строку line
vector<string> vs;
string word;
while (ss>>word)      // считываем слова без знаков пунктуации
      vs.push_back(word);
```

Применив такой способ, получаем желаемый результат.

```
As
planned
the
guests
arrived
then
```

К сожалению, этот код слишком сложен и излишне специализирован. А что делать, если знаки пунктуации определены иначе? Опишем более общий и полезный способ удаления нежелательных символов из потока ввода. Как должен выглядеть этот поток? Как должен выглядеть наш код? Может быть, так?

```
ps.whitespace(";:,."); // точка с запятой, двоеточие, запятая и точка
                       // считаются разделителями
string word;
while (ps>>word) vs.push_back(word);
```

Как определить поток, работающий так, как поток **ps**? Основная идея заключается в том, чтобы считывать слова в обычный поток ввода, а затем обрабатывать символы-разделители, заданные пользователем, как настоящие разделители, т.е. не передавать разделители пользователю, а просто использовать их для отделения слов друг от друга. Рассмотрим пример.

```
as.not
```

Слова **as** и **not** должны быть двумя самостоятельными словами

```
as
not
```

Для того чтобы сделать это, можно определить класс. Он должен принимать символы из потока **istream** и содержать оператор **>>**, работающий так же, как оператор ввода потока **istream**, за исключением того, что мы сами можем указывать, какие символы являются разделителями. Для простоты будем считать существующие символы-разделители (пробел, символ перехода на новую строку и т.д.) обыч-

ными символами; мы просто позволим пользователю указать дополнительные разделители. Кроме того, мы не будем удалять указанные символы из потока; как и прежде, мы превратим их в разделители. Назовем наш класс **Punct_stream**.

```
class Punct_stream { // аналогичен потоку istream, но пользователь
                     // может самостоятельно задавать разделители
public:
     Punct_stream(istream& is)
         : source(is), sensitive(true) { }

     void whitespace(const string& s) // создает строку
                                      // разделителей s
         { white = s; }
     void add_white(char c) { white += c; } // добавляет символ
                                            // в набор разделителей
     bool is_whitespace(char c); // является ли с набором
                                 // разделителей?

     void case_sensitive(bool b) { sensitive = b; }
     bool is_case_sensitive() { return sensitive; }

     Punct_stream& operator>>(string& s);
     operator bool();
private:
     istream& source;          // источник символов
     istringstream buffer;     // буфер для форматирования
     string white;             // символы-разделители
     bool sensitive;           // является ли поток чувствительным
                               // к регистру?
};
```

Как и в предыдущем примере, основная идея — ввести строку из потока **istream** как одно целое, преобразовать символы-разделители в пробелы, а затем использовать поток **stringstream** для форматирования. Кроме обработки разделителей, заданных пользователем, в классе **Punct_stream** есть аналогичная возможность: если вызвать функцию **case_sensitive()**, то она преобразует ввод, чувствительный к регистру, в нечувствительный.

Например, можно приказать объекту класса **Punct_stream** прочитать строку

Man bites dog!

как

man
bites
dog

Конструктор класса **Punct_stream** получает поток **istream**, используемый как источник символов, и присваивает ему локальное имя **source**. Кроме того, конструктор по умолчанию делает поток чувствительным к регистру, как обычно. Можно создать объект класса **Punct_stream**, считывающий данные из потока **cin**, рас-

сматривающий точку с запятой, двоеточие и точку как разделители, а также переводящий все символы в нижний регистр.

```
Punct_stream ps(cin);      // объект ps считывает данные из потока cin
ps.whitespace(";:.");      // точка с запятой, двоеточие и точка
                           // также являются разделителями
ps.case_sensitive(false); // нечувствительный к регистру
```

Очевидно, что наиболее интересной операцией является оператор ввода **>>**. Он также является самым сложным для определения. Наша общая стратегия состоит в том, чтобы считать всю строку из потока **istream** в строку **line**. Затем мы превратим все наши разделители в пробелы (**' '**). После этого отправим строку в поток **istringstream** с именем **buffer**. Теперь для считывания данных из потока **buffer** можно использовать обычные разделители и оператор **>>**. Код будет выглядеть немного сложнее, поскольку мы только пытаемся считать данные из потока **buffer** и заполняем его, только если он пуст.

```
Punct_stream& Punct_stream::operator>>(string& s)
{
     while (!(buffer>>s)) { // попытка прочитать данные
                            // из потока buffer
          if (buffer.bad() || !source.good()) return *this;
          buffer.clear();

          string line;
          getline(source,line); // считываем строку line
                                // из потока source

          // при необходимости заменяем символы
          for (int i =0; i<line.size(); ++i)
               if (is_whitespace(line[i]))
                  line[i]= ' ';                // в пробел
               else if (!sensitive)
                    line[i] = tolower(line[i]); // в нижний регистр

          buffer.str(line); // записываем строку в поток
     }
     return *this;
}
```

Рассмотрим этот код шаг за шагом. Сначала обратим внимание не нечто необычное.

```
while (!(buffer>>s)) {
```

Если в потоке **buffer** класса **istringstream** есть символы, то выполняется инструкция **buffer>>s** и объект **s** получит слово, разделенное разделителями; больше эта инструкция ничего не делает. Эта инструкция будет выполняться, пока в объекте **buffer** есть символы для ввода. Однако, когда инструкция **buffer>>s** не сможет выполнить свою работу, т.е. если выполняется условие **!(buffer>>s)**, мы должны

наполнить объект **buffer** символами из потока **source**. Обратите внимание на то, что инструкция **buffer>>s** выполняется в цикле; после попытки заполнить объект **buffer** мы должны снова попытаться выполнить ввод.

```
while (!(buffer>>s)) { // попытка прочитать символы из буфера
     if (buffer.bad() || !source.good()) return *this;
     buffer.clear();

     // заполняем объект buffer
}
```

Если объект **buffer** находится в состоянии **bad()** или существуют проблемы с источником данных, работа прекращается; в противном случае объект **buffer** очищается и выполняется новая попытка. Мы должны очистить объект **buffer**, потому что попадем в "цикл заполнения", только если попытка ввода закончится неудачей. Обычно это происходит, если вызывается функция **eof()** для объекта **buffer**; иначе говоря, когда в объекте **buffer** не остается больше символов для чтения. Обработка состояний потока всегда запутанна и часто является причиной очень тонких ошибок, требующих утомительной отладки. К счастью, остаток цикла заполнения вполне очевиден.

```
string line;
getline(source,line); // вводим строку line из потока source

// при необходимости выполняем замену символов
for (int i =0; i<line.size(); ++i)
     if (is_whitespace(line[i]))
        line[i]= ' ';                // в пробел
     else if (!sensitive)
        line[i] = tolower(line[i]); // в нижний регистр
buffer.str(line);                    // вводим строку в поток
```

Считываем строку в объект **buffer**, затем просматриваем каждый символ строки в поисках кандидатов на замену. Функция **is_whitespace()** является членом класса **Punct_stream**, который мы определим позднее. Функция **tolower()** — это стандартная библиотечная функция, выполняющая очевидное задание, например превращает символ **A** в символ **a** (см. раздел 11.6).

После правильной обработки строки **line** ее необходимо записать в поток **istringstream**. Эту задачу выполняет функция **buffer.str(line)**; эту команду можно прочитать так: "Поместить строку из объекта **buffer** класса **stringstream** в объект **line**".

Обратите внимание на то, что мы "забыли" проверить состояние объекта **source** после чтения данных с помощью функции **getline()**. Это не обязательно, поскольку в начале цикла выполняется проверка условия **!source.good()**.

Как всегда, оператор **>>** возвращает ссылку на поток ***this** (раздел 17.10).

Проверка разделителей проста; мы сравниваем символ с каждым символом из строки, в которой записаны разделители.

```
bool Punct_stream::is_whitespace(char c)
{
     for (int i = 0; i<white.size(); ++i)
          if (c==white[i]) return true;
     return false;
}
```

Напомним, что поток **istringstream** обрабатывает обычные разделители (например, символы перехода на новую строку или пробел) по-прежнему, поэтому никаких особых действий предпринимать не надо.

Осталась одна загадочная функция.

```
Punct_stream::operator bool()
{
     return !(source.fail() || source.bad()) && source.good();
}
```

Обычное использование потока **istream** сводится к проверке результата оператора **>>**. Рассмотрим пример.

```
while (ps>>s) { /* . . . */ }
```

Это значит, что нам нужен способ для проверки результата выполнения инструкции **ps>>s**, представленного в виде булевого значения. Результатом инструкции **ps>>s** является объект класса **Punct_stream**, поэтому нам нужен способ неявного преобразования класса **Punct_stream** в тип **bool**. Эту задачу решает функция **operator bool()** в классе **Punct_stream**.

Функция-член **operator bool()** определяет преобразование класса **Punct_stream** в тип **bool**. В частности, она возвращает значение **true**, если эта операция над классом **Punct_stream** прошла успешно.

Теперь можем написать программу.

```
int main()
     // вводит текст и создает упорядоченный список всех слов
     // из заданного текста, игнорируя знаки пунктуации и регистры,
     // а также удаляя дубликаты из полученного результата
{
     Punct_stream ps(cin);
     ps.whitespace(";:,.?!()\"{}<>/&$@#%^*|~"); // \" в строке
                                                // означает "
     ps.case_sensitive(false);

     cout << "Пожалуйста, введите слова\n";
     vector<string> vs;
     string word;
     while (ps>>word) vs.push_back(word); // ввод слов
```

```
    sort(vs.begin(),vs.end()); // сортировка в лексикографическом
                               // порядке
    for (int i=0; i<vs.size(); ++i) // запись в словарь
        if (i==0 || vs[i]!=vs[i-1]) cout << vs[i] << endl;
}
```

Этот код создает упорядоченный список введенных слов. Инструкция

```
if (i==0 || vs[i]!=vs[i-1])
```

удаляет дубликаты. Если в программу ввести слова

```
There are only two kinds of languages: languages that people complain
about, and languages that people don't use.
```

то результат ее работы будет выглядеть следующим образом:

```
and
are
complain
don't
languages
of
only
people
that
there
two
use
```

Почему мы получили на выходе **don't**, а не **dont**? Потому что оставили апостроф за пределами списка разделителей **whitespace()**.

Внимание: класс **Punct_stream** во многом похож на класс **istream**, но на самом деле отличается от него. Например, мы не можем проверить его состояние с помощью функции **rdstate()**, функция **eof()** не определена, и нет оператора **>>**, который вводит целые числа. Важно отметить, что мы не можем передать объект класса **Punct_stream** в качестве аргумента функции, ожидающей поток **istream**. Можно ли определить класс **Punct_istream**, который в точности повторял бы поведение класса **istream**? Можно, но у вас пока нет достаточного опыта программирования, вы еще не освоили основы проектирования и не знаете всех возможностей языка (если впоследствии вы вернетесь к этой задаче, то сможете реализовать буферы потоков на уровне профессионала).

Легко ли читать определение класса **Punct_stream**? Понятны ли вам объяснения? Могли бы вы самостоятельно написать такую программу? Еще несколько дней назад вы были новичком и честно закричали бы: “Нет, нет! Никогда!” или “Нет, нет! Вы что, с ума сошли? Очевидно, что ответ на поставленный вопрос отрицательный”. Цель нашего примера заключается в следующем:

- показать реальную задачу и способ ее решения;
- доказать, что это решение можно найти с помощью вполне доступных средств;

- описать простое решение простой задачи;
- продемонстрировать разницу между интерфейсом и реализацией.

Для того чтобы стать программистом, вы должны читать программы, причем не только учебные. Приведенный пример относится как раз к таким задачам. Через несколько дней или недель вы разберетесь в нем без труда и сможете улучшить это решение.

Этот пример можно сравнить с уроком, на котором учитель английского языка для иностранцев произносит выражения на сленге, чтобы показать его колорит и живость.

11.8. И еще много чего

Подробности ввода-вывода можно описывать бесконечно. Этот процесс ограничен лишь терпением слушателей. Например, мы не рассмотрели сложности, связанные с естественными языками. То, что в английском языке записывается как **`12.35`**, в большинстве европейских языков означает **`12,35`**. Естественно, стандартная библиотека C++ предоставляет возможности для устранения этих и многих других проблем. А как записать китайские иероглифы? Как сравнивать строки, записанные символами малайского языка? Ответы на эти вопросы существуют, но они выходят далеко за рамки нашей книги. Если вам потребуется более детальная информация, можете обратиться к более специализированным книгам (например, Langer, *Standard C++ IOStreams and Locales* и Stroustrup, *The C++ Programming Language*), а также к библиотечной и системной документации. Ищите ключевое слово *locale* (местная специфика); этот термин обычно применяется к функциональным возможностям для обработки различий между естественными языками.

Другим источником сложностей является буферизация; стандартные библиотечные потоки **`iostream`** основаны на концепции под названием **`streambuf`**. Для сложных задач, связанных с потоками **`iostream`**, при решении которых важна производительность или функциональность, без объектов класса **`istreambuf`** обойтись нельзя. Если хотите определить свой собственный класс **`iostream`** или настроить объекты класса **`iostream`** на новые источники данных, см. главу 21 книги *The C++ Programming Language* Страуструпа или системную документацию.

При программировании на языке C++ вы можете обнаружить семейство стандартных функций ввода-вывода **`printf()`**/**`scanf()`**, определенных в языке C. В этом случае прочитайте разделы 27.6, B.10.2, или прекрасный учебник Кернигана и Ритчи *Язык программирования C* (Kernighan and Ritchie, *The C Programming Language*), или же любой из многочисленных источников информации в веб. Каждый язык имеет свои собственные средства ввода-вывода; все они изменяются, иногда неправильно, но в большинстве случаев правильно (совершенно по-разному) отражая основные понятия, изложенные в главах 10 и 11.

Стандартная библиотека ввода-вывода описана в приложении Б, а связанные с ней графические пользовательские интерфейсы — в главах 12–16.

Задание

1. Напишите программу с именем **Test_output.cpp**. Объявите целочисленную переменную **birth_year** и присвойте ей год своего рождения.
2. Выведите переменную **birth_year** в десятичном, шестнадцатеричном и восьмеричном виде.
3. Выведите основание системы счисления для каждого числа.
4. Выровняли ли вы результаты по столбцам с помощью символа табуляции? Если нет, то сделайте это.
5. Теперь выведите год вашего рождения.
6. Были ли какие-то проблемы? Что произошло? Замените ваш вывод на десятичный.
7. Вернитесь к упр. 2 и выведите основание системы счисления для каждого числа.
8. Попытайтесь прочитать данные как восьмеричные, шестнадцатеричные и т.д.

```
cin >> a >>oct >> b >> hex >> c >> d;
cout << a << '\t'<< b << '\t'<< c << '\t'<< d << '\n' ;
```

   Запустите программу со следующими входными данными:

```
1234 1234 1234 1234
```

   Объясните результаты.

9. Напишите программу, три раза выводящую на печать число **1234567.89**: сначала в формате **general**, затем — в **fixed** и в **scientific**. Какой способ представления обеспечивает наибольшую точность? Почему?
10. Создайте простую таблицу, содержащую фамилию, имя, телефонный номер и адрес электронной почты хотя бы пяти ваших друзей. Поэкспериментируйте с разной шириной полей, пока не найдете приемлемый.

Контрольные вопросы

1. Почему ввод-вывод является сложной задачей для программиста?
2. Что означает выражение **<< hex**?
3. Какие шестнадцатеричные числа используются в компьютерных науках? Почему?
4. Перечислите несколько возможностей, которые вы хотели бы реализовать при форматировании вывода целых чисел.
5. Что такое манипулятор?
6. Назовите префикс десятичного, восьмеричного и шестнадцатеричного числа.
7. Какой формат по умолчанию применяется при выводе чисел с плавающей точкой?
8. Что такое поле вывода?
9. Объясните, что делают функции **setprecision()** и **setw()**.

10. Для чего нужны разные режимы при открытии файлов?
11. Какие из перечисленных далее манипуляторов не являются инертными: **hex**, **scientific**, **setprecision**, **showbase**, **setw**?
12. Укажите разницу между символьным и двоичным вводом.
13. Приведите пример, демонстрирующий преимущество использования двоичного файла вместо текстового.
14. Приведите два примера, в которых может оказаться полезным класс **stringstream**.
15. Что такое позиция в файле?
16. Что произойдет, если позиция в файле будет установлена за его пределами?
17. Когда ввод строк предпочтительнее, чем ввод, ориентированный на тип?
18. Что делает функция **isalnum(c)**?

Термины

fixed	восьмеричный	нестандартный разделитель
general	двоичный	позиционирование файла
noshowbase	десятичный	регулярность
scientific	классификация символов	строковый ввод
setprecision	манипулятор	форматирование вывода
showbase	нерегулярность	шестнадцатеричный

Упражнения

1. Напишите программу, вводящую текстовый файл и записывающую его содержимое в новый файл, используя нижний регистр.
2. Напишите программу, удаляющую из файла все гласные буквы. Например, фраза **Once upon a time!** принимает вид **nc pn tm!**. Удивительно часто результат остается вполне читабельным; проверьте это на своих друзьях.
3. Напишите программу под названием **multi_input.cpp**, которая предлагает пользователю ввести несколько целых восьмеричных, десятичных и шестнадцатеричных чисел в любом сочетании, используя суффиксы **0** и **0x**; интерпретируйте эти числа правильно и приведите в десятичный вид. Ваша программа должна выводить на экран примерно такие результаты:

```
0x4 шестнадцатеричное превращается в 67 десятичное
0123 восьмеричное превращается в 83 десятичное
65 десятичное превращается в 65 десятичное
```

4. Напишите программу, считывающую строки и выводящую категории каждого символа в соответствии с правилами классификации, описанными в разделе 11.6. Помните, что один и тот же символ может относиться к разным категориям (например, **x** — это и буквенный, и буквенно-цифровой символ).

5. Напишите программу, заменяющую знаки пунктуации пробелами. Например, строка “ **- don’t use the as-if rule**”. принимает вид “ **dont use the asif rule** ”.
6. Модифицируйте программу из предыдущего упражнения, чтобы она заменяла сокращения **don’t** словами **do not**, **can’t — cannot** и т.д.; дефисы внутри слов не трогайте (таким образом, мы получим строку “ **do not use the as-if rule**”); переведите все символы в нижний регистр.
7. Используйте программу из предыдущего упражнения для создания словаря (в качестве альтернативы подходу, описанному в разделе 11.7). Примените ее к многостраничному текстовому файлу, проанализируйте результат и подумайте, можно ли улучшить эту программу, чтобы получить более качественный словарь.
8. Разделите программы ввода-вывода из раздела 11.3.2 на две части: одна программа пусть конвертирует обычный текстовый файл в двоичный, а другая — считывает двоичный файл и преобразует его в текстовый. Протестируйте эти программы, сравнивая текстовые файлы до и после преобразования в двоичный файл.
9. Напишите функцию **vector<string> split(const string& s)**, возвращающую вектор подстрок аргумента **s**, разделенных пробелами.
10. Напишите функцию **vector<string> split(const string& s, const string& w)**, возвращающую вектор подстрок аргумента **s**, между которыми стоят разделители, при условии, что в качестве разделителя может использоваться как обычный пробел, так и символы из строки **w**.
11. Измените порядок следования символов в текстовом файле. Например, строка **asdfghjkl** примет вид **lkjhgfdsa**. Подсказка: вспомните о режимах открытия файлов.
12. Измените порядок следования слов (определенных как строки, разделенные пробелами). Например, строка **Norwegian Blue parrot** примет вид **parrot Blue Norwegian**. Вы можете предположить, что все строки из файла могут поместиться в памяти одновременно.
13. Напишите программу, считывающую текстовый файл и записывающую в другой файл количество символов каждой категории (см. раздел 11.6).
14. Напишите программу, считывающую из файла числа, разделенные пробелами, и выводящую в другой файл числа, используя научный формат и точность, равную восьми в четырех полях по двадцать символов в строке.
15. Напишите программу, считывающую из файла числа, разделенные пробелами, и выводящую их в порядке возрастания по одному числу в строке. Каждое число должно быть записано только один раз, если обнаружится дубликат, то необходимо вывести количество таких дубликатов в строке. Например, срока “**7 5 5 7 3 117 5**” примет следующий вид:

```
3
5 3
7 2
117
```

Послесловие

Ввод и вывод сложны, поскольку вкусы и предпочтения у людей разные и не подчиняются стандартизации и математическим законам. Как программисты мы редко имеем право навязывать пользователям свои взгляды, а когда можем выбирать, должны сдерживаться и стараться предлагать простые альтернативы, которые выдержат проверку временем. Следовательно, мы должны смириться с определенными неудобствами ввода и вывода и стремиться, чтобы наши программы были как можно более простыми, но не проще.