8

Символьный ввод-вывод и проверка достоверности ввода

В ЭТОЙ ГЛАВЕ...

- Дополнительные сведения о вводе, выводе и различия между буферизированным и небуферизированным вводом
- Моделирование условия конца файла с клавиатуры
- Использование перенаправления для подключения программы к файлам
- Создание более дружественных пользовательских интерфейсов

В мире вычислений мы используем слова *ввод* и *вывод* многими путями. Мы обсуждаем устройства ввода и вывода, такие как клавиатуры, устройства USB, сканеры и лазерные принтеры. Мы говорим о данных, применяемых для ввода и вывода. Мы упоминаем функции, которые выполняют ввод и вывод. В этой главе основное внимание уделяется функциям ввода-вывода.

Функции ввода-вывода перемещают информацию в программу и из нее; примерами могут быть `printf()`, `scanf()`, `getchar()` и `putchar()`. Вы уже сталкивались с этими функциями в предшествующих главах, а теперь вы ознакомитесь с концепциями, лежащими в их основе. Наряду с этим вы увидите, как улучшить интерфейс между пользователем и программой.

Первоначально функции ввода-вывода не были частью определения языка С. Их разработка была оставлена за реализациями. На практике моделью для этих функций служила реализация С для операционной системы Unix. Учитывая весь прежний опыт, библиотека ANSI С содержит большое количество таких функций ввода-вывода, ориентированных на Unix, включая те, что мы использовали ранее. Поскольку эти стандартные функции должны работать с широким разнообразием компьютерных сред, они редко извлекают преимущества из возможностей, присущих конкретной системе. Поэтому многие поставщики реализаций языка С предлагают дополнительные функции ввода-вывода, которые задействуют специальные средства оборудования. Другие функции или семейства функций включаются в отдельные операционные системы, которые поддерживают, например, специальные графические интерфейсы вроде предоставляемых в Windows и Macintosh. Эти специализированные нестандартные функции позволяют писать программы, которые эксплуатируют конкретный компьютер более эффективно. К сожалению, часто они не могут применяться в других компьютерных системах.

Таким образом, мы сосредоточимся на стандартных функциях ввода-вывода, доступных для всех систем, т.к. они позволяют разрабатывать переносимые программы, которые можно легко перемещать из одной системы в другую. Они также стимулируют использование в программах файлов для ввода и вывода.

Многие программы сталкиваются с одной важной задачей — проверкой допустимости входных данных, т.е. с выяснением, ввел ли пользователь данные, которые ожидаются программой. В этой главе рассматриваются некоторые проблемы и решения, связанные с проверкой допустимости вводимых данных.

Односимвольный ввод-вывод: `getchar()` и `putchar()`

Как было показано в главе 7, функции `getchar()` и `putchar()` выполняют ввод и вывод по одному символу за раз. Такой подход может показаться нерациональным. В конце концов, можно легко читать группы, состоящие из нескольких символов, но этот метод вполне вписывается в возможности компьютера. Более того, такой подход лежит в основе большинства программ, имеющих дело с текстом — т.е. с обычными словами. Чтобы напомнить, как работают эти функции, в листинге 8.1 приведен очень простой пример. Здесь всего лишь принимаются символы из клавиатурного ввода и затем отображаются на экране. Такой процесс называется *эхо-выводом ввода*. В коде применяется цикл `while`, который завершается при обнаружении символа `#`.

Листинг 8.1. Программа `echo.c`

```
/* echo.c -- повторяет ввод */
#include <stdio.h>
int main(void)
{
   char ch;
   while ((ch = getchar()) != '#')
      putchar(ch);
      return 0;
}
```

Со времен появления стандарта ANSI C в языке с использованием функций `getchar()` и `putchar()` ассоциирован заголовочный файл `stdio.h`, потому он и был включен в программе. (Обычно `getchar()` и `putchar()` не являются истинными функциями, а определены с применением макросов препроцессора, как будет раскрыто в главе 16.) Выполнение программы приводит к обмену следующего вида:

```
Здравствуйте. Я хотел бы[enter]
Здравствуйте. Я хотел бы
приобрести #3 мешка картофеля.[enter]
приобрести
```

После наблюдения за работой этой программы может возникнуть вопрос, почему нужно набирать строку полностью, прежде чем введенные символы будут повторены на экране. Вас также может интересовать, есть ли лучший способ завершения ввода. Использование для завершения ввода специального символа, такого как #, предотвращает его употребление в тексте. Чтобы ответить на эти вопросы, давайте посмотрим, каким образом программы на языке C обрабатывают клавиатурный ввод. В частности, мы выясним, что собой представляет буферизация, и ознакомимся с понятием стандартного входного файла.

Буферы

Если вы запустите предыдущую программу на некоторых более старых системах, то вводимый текст может отображаться на экране немедленно. То есть выполнение этой программы могло бы дать примерно такой результат:

```
ЗЗддррааввссттввуужйттее..  ЯЯ  ххооттеелл  ббыы[enter]
ппррииооббрреессттии  #
```

Такое поведение является исключением. В большинстве систем ничего не произойдет до тех пор, пока не будет нажата клавиша <Enter>, как в первом примере запуска. Немедленный эхо-вывод вводимых символов на экране представляет собой пример *небуферизированного* (или *прямого*) ввода, при котором набираемые символы немедленно становятся доступным для ожидающей их программы. С другой стороны, задержанный эхо-вывод иллюстрирует *буферизированный* ввод, когда введенные символы накапливаются и хранятся во временной области, называемой *буфером*. Нажатие клавиши <Enter> делает блок набранных символов доступным для программы. Эти две разновидности ввода сравниваются на рис. 8.1.

Зачем иметь буферы? Во-первых, передача нескольких символов в виде блока является менее затратной по времени, чем отправка символов по одному. Во-вторых, в случае опечатки можно воспользоваться средствами корректировки, поддерживаемыми клавиатурой, и исправить опечатку. Затем после финального нажатия <Enter> программе будет передана исправленная версия.

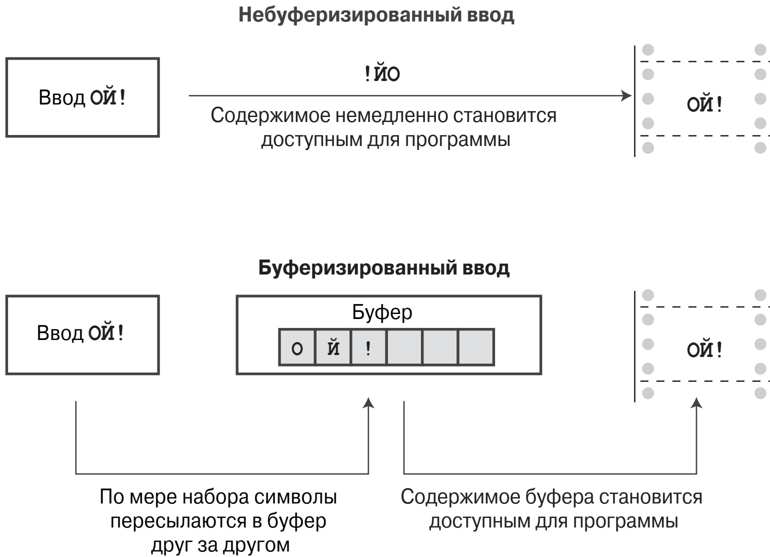

Рис. 8.1. *Буферизированный и небуферизированный ввод*

С другой стороны, небуферизированный ввод желателен для некоторых интерактивных программ. Например, в играх требуется, чтобы каждая команда выполнялась сразу же после нажатия клавиши. Таким образом, и буферизированный, и небуферизированный ввод имеют свои применения.

Существуют два вида буферизации — *полностью буферизированный* ввод-вывод и *построчно буферизированный* ввод-вывод. При полностью буферизированном вводе буфер сбрасывается (его содержимое отправляется в место назначения), когда он полон. Буферизация такого вида обычно происходит при файловом вводе. Размер буфера зависит от системы, но наиболее распространены значения 512 и 4096 байтов. В случае построчно буферизированного ввода-вывода буфер сбрасывается всякий раз, когда появляется символ новой строки. Клавиатурный ввод обычно является построчно буферизированным, так что нажатие <Enter> вызывает сброс буфера.

Каким типом ввода вы располагаете — буферизированным или небуферизированным? В ANSI C и последующих стандартах C указано, что ввод должен быть буферизированным, но в K&R C выбор изначально возлагался на разработчика компилятора. Тип ввода, используемый в системе, можно определить, запустив на выполнение программу `echo.c` и проанализировав ее поведение.

Причина того, что в ANSI C было принято решение считать стандартом буферизированный ввод, связана с тем, что некоторые компьютерные системы не разрешают небуферизированный ввод. Если ваш компьютер допускает небуферизированный ввод, то весьма вероятно, что применяемый вами компилятор C предлагает небуферизированный ввод в качестве опции. Например, многие компиляторы для компьютеров, совместимых с IBM PC, предоставляют специальное семейство функций, которые поддерживаются заголовочным файлом `conio.h` и предназначены для небуферизированного ввода. К их числу относятся функция `getche()` для небуферизированного ввода с эхо-выводом и функция `getch()` для небуферизированного ввода без эхо-вывода. (*Ввод с эхо-выводом* означает, что вводимый символ отображается на экране, а *ввод без эхо-вывода* — что нажатия клавиш не приводят отображению символов на экране.) В системах Unix используется другой подход, при котором буферизацией управляет сама система

Unix. В Unix вы применяете функцию `ioctl()` (которая входит в состав библиотеки Unix, но не является частью стандарта С) для указания желаемого типа ввода, после чего функция `getchar()` ведет себя должным образом. В ANSI С функции `setbuf()` и `setvbuf()` (глава 13) предоставляют определенный контроль над буферизацией, но присущие ряду систем ограничения снижают их эффективность. Выражаясь кратко, не существует способа, соответствующего стандарту ANSI, для обеспечения небуферизированного ввода; такие средства зависят от компьютерной системы. В этой книге мы предполагаем, что вы используете буферизированный ввод.

Завершение клавиатурного ввода

Программа `echo.c` останавливается, когда введен символ #, что удобно до тех пор, пока этот символ исключен из обычных входных данных. Однако, как уже было показано, символ # может встречаться и в обычном вводе. В идеальном случае хотелось бы иметь символ завершения ввода, который в обычном тексте не встречается. Такой символ не может неожиданно появиться в середине входного текста, останавливая программу раньше, чем планировалось. В С имеется ответ на эту потребность, но чтобы понять его, необходимо знать, как в С работать с файлами.

Файлы, потоки и ввод данных с клавиатуры

Файл — это область памяти, в которой хранится информация. Обычно файл размещается в постоянной памяти определенного вида, такого как жесткий диск, флэш-накопитель USB или оптический диск вроде DVD. Важность файлов для компьютерных систем не вызывает сомнений. Например, ваши программы на С хранятся в файлах, то же самое можно сказать о программах, применяемых для компиляции этих программ. Последний пример указывает на то, что некоторым программам требуется возможность доступа к отдельным файлам. При компиляции программы, хранящейся в файле `echo.c`, компилятор открывает этот файл и читает его содержимое. После завершения компиляции файл закрывается. Другие программы, такие как текстовые процессоры, не только открывают, читают и закрывают файлы, но также и записывают в них.

В С, как у мощного, гибкого и т.д. языка, имеется много библиотечных функций, предназначенных для открытия, чтения, записи и закрытия файлов. На одном уровне он может иметь дело с файлами, используя для этого базовые инструменты для работы с файлами из операционной системы. Это называется *низкоуровневым вводом-выводом*. Из-за многочисленных отличий между компьютерными системами создать стандартную библиотеку универсальных функций для низкоуровневого ввода-вывода невозможно, и в стандарте ANSI С такая попытка даже не предпринимается. Тем не менее, язык С также умеет работать с файлами на другом уровне, который имеет название *стандартный пакет ввода-вывода*. При этом предполагается создание стандартной модели и стандартного набора функций ввода-вывода, предназначенных для работы с файлами. На таком более высоком уровне различия между системами поддерживаются специфическими реализациями С, так что вы имеете дело с унифицированным интерфейсом.

А о каких отличиях между компьютерными системами идет речь? Например, разные системы сохраняют файлы по-разному. Некоторые хранят содержимое файла в одном месте, а информацию о нем — в другом. Одни системы встраивают описание файла в сам файл. При работе с текстами многие системы для обозначения конца строки применяют одиночный символ новой строки. Другие могут использовать для этого комбинацию символов возврата каретки и перевода строки. Некоторые системы измеряют размер файлов до ближайшего байта, а другие — в блоках байтов.

Когда вы применяете стандартный пакет ввода-вывода, вы защищены от воздействия таких отличий. Следовательно, для проверки на предмет символа новой строки можно использовать конструкцию `if (ch == '\n')`. Если система применяется комбинация символов возврата каретки и перевода строки, то функции ввода-вывода выполняют автоматическую трансляцию между двумя этими представлениями в обоих направлениях.

Концептуально программа на C имеет дело с потоком, а не напрямую с файлом. *Поток* — это идеализированное течение данных, на которое отображается действительный ввод или вывод. Это означает, что разнообразные виды ввода с отличающимися свойствами представлены с помощью потоков, имеющих более унифицированные свойства. Тогда процесс открытия файла становится процессом ассоциирования потока с файлом, а чтение и запись осуществляются через поток.

Файлы подробно обсуждаются в главе 13. Для целей настоящей главы просто запомните, что в языке C устройства ввода и вывода трактуются таким же образом, как обычные файлы на устройствах хранения. В частности, клавиатура и устройство отображения считаются файлами, автоматически открываемыми каждой программой на C. Клавиатурный ввод представлен потоком по имени `stdin`, а вывод на экран (или на телетайп либо другое устройство вывода) представлен потоком по имени `stdout`. Функции `getchar()`, `putchar()`, `printf()` и `scanf()` являются членами стандартного пакета ввода-вывода, и все они имеют дело с двумя упомянутыми потоками.

Одно из следствий всего этого заключается в том, что при работе с клавиатурным вводом можно использовать те же самые приемы, как и при работе с файлами. Например, программе, читающей файл, необходим способ обнаружения конца файла, чтобы знать, где останавливать чтение. Поэтому функции для ввода в C оснащены встроенным средством обнаружения конца файла. Поскольку клавиатурный ввод трактуется подобно файлу, вы должны иметь возможность применять это средство обнаружения конца файла также и в данном случае. Давайте посмотрим, как это делается, начав с файлов.

Конец файла

Операционная система нуждается в каком-то способе для выяснения, где начинается и где заканчивается каждый файл. Один из методов обнаружения конца файла предусматривает помещение в файл специального символа, помечающего его конец. В свое время такой метод использовался, к примеру, в текстовых файлах в средах операционных систем CP/M, IBM-DOS и MS-DOS. Теперь эти операционные системы для пометки конца файла могли бы применять встроенный символ <Ctrl+Z>. Когда-то это было единственным средством, которое использовали операционные системы, но сейчас доступны другие варианты наподобие отслеживания размера файла. Таким образом, современный текстовый файл может содержать, а может и не содержать встроенный символ <Ctrl+Z>, однако если он присутствует, операционная система будет трактовать его как маркер конца файла. Этот подход иллюстрируется на рис. 8.2.

Фраза:

Робот Бишоп
плавно открыл люк
и ответил на свой вызов.

Фраза в файле:

```
Робот Бишоп\nплавно открыл люк\nи ответил на свой вызов.\n^Z
```

***Рис. 8.2.** Файл с маркером конца файла*

Второй подход заключается в том, что операционная система хранит информацию о размере файла. Если файл содержит 3000 байтов, а программа прочитала 3000 байтов, значит, она достигла конца файла. Операционная система MS-DOS и ей подобные применяют этот подход для двоичных файлов, т.к. данный метод позволяет хранить в файлах все символы, в том числе <Ctrl+Z>. Более новые версии DOS также используют этот подход для текстовых файлов. В Unix он применяется ко всем файлам.

Такое многообразие методов поддерживается в C за счет того, что функция `getchar()` возвращает специальное значение при достижении конца файла независимо от того, как в действительности конец файла обнаруживается операционной системой. Этому специальному значению было назначено имя `EOF` (“end of file” — “конец файла”). Следовательно, возвращаемым значением функции `getchar()` в случае обнаружения конца файла является `EOF`. Функция `scanf()` при обнаружении конца файла также возвращает `EOF`. Обычно `EOF` определяется в файле `stdio.h` следующим образом:

```
#define EOF (-1)
```

Почему было выбрано значение `-1`? Обычно функция `getchar()` возвращает значение в диапазоне от `0` до `127`, поскольку они соответствуют стандартному набору символов, но она может возвращать значения от `0` до `255`, если система распознает расширенный набор символов. В любом случае значение `-1` не относится ни к одному из символов, так что оно может использоваться для сообщения о конце файла.

В некоторых системах `EOF` может быть определено как значение, не равное `-1`, но это определение всегда отличается от возвращаемого значения, генерируемого допустимым входным символом. Если вы включили файл `stdio.h` и применяете символическую константу `EOF`, то не обязаны беспокоиться о ее числовом определении. Важно понимать, что `EOF` представляет значение, сигнализирующее об обнаружении конца файла, а не значение, действительно находящееся файле.

А как использовать `EOF` в программе? Нужно сравнить возвращаемое значение `getchar()` с `EOF`. Если они отличаются, конец файла пока еще не достигнут. Другими словами, можно указывать выражение, подобное следующему:

```
while ((ch = getchar()) != EOF)!
```

Что, если производится чтение клавиатурного ввода, а не файла? Большинство систем (но не все) поддерживают какой-то способ эмулировать условие конца файла с помощью клавиатуры. С учетом этого базовую программу чтения и эхо-вывода можно переписать, как показано в листинге 8.2.

Листинг 8.2. Программа `echo_eof.c`

```
/* echo_eof.c — повторяет на экране ввод до достижения конца файла */
#include <stdio.h>
int main(void)
{
   int ch;

   while ((ch = getchar()) != EOF)
      putchar(ch);

   return 0;
}
```

Обратите внимание на перечисленные ниже аспекты.

- Вам не придется определять EOF, т.к. об этом уже позаботился заголовочный файл stdio.h.
- Вам не нужно беспокоиться о действительном значении EOF, поскольку оператор #define в файле stdio.h позволяет иметь дело с символическим представлением EOF. Вы не должны писать код, в котором предполагается, что EOF имеет какое-то конкретное значение.
- Тип переменной ch изменен с char на int, потому что переменные типа char могут быть представлены целочисленными значениями без знака в диапазоне от 0 до 255, но EOF может иметь числовое значение -1. Такое значение недопустимо для типа char без знака, но допустимо для типа int. К счастью, функция getchar() сама имеет тип int, поэтому она может читать символ EOF. В реализациях, поддерживающих тип char со знаком, можно обойтись объявлением переменной ch как имеющей тип char, но лучше применить более общую форму.
- Именно из-за того, что функция getchar() имеет тип int, некоторые компиляторы предупреждают о возможной потере данных при присваивании возвращаемого значения этой функции переменной типа char.
- Тот факт, что переменная ch является целочисленной, никак не беспокоит функцию putchar(). Она по-прежнему выводит символьный эквивалент.
- Чтобы использовать эту программу с клавиатурным вводом, необходим какой-то способ набора символа EOF. Понятно, что нельзя просто набрать буквы *E O F* или ввести -1. (Ввод с клавиатуры -1 приводит к передаче в программу двух символов: дефиса и цифры 1.) Взамен понадобится выяснить, какой символ требует система. Например, в большинстве систем Unix и Linux нажатие комбинации клавиш <Ctrl+D> в *начале* строки вызывает передачу сигнала конца файла. Многие системы в качестве сигнала конца файла распознают комбинацию <Ctrl+Z> в начале строки, а некоторые интерпретируют ее как таковую в любом месте строки.

Ниже показан пример применения буферизированного ввода в программе echo_eof.c под управлением Unix:

```
У него не было даже пальто.
У него не было даже пальто.
 В город молодой человек вошел в зеленом в талию костюме.
 В город молодой человек вошел в зеленом в талию костюме.
                   И. Ильф, Е. Петров
                   И. Ильф, Е. Петров
[Ctrl+D]
```

Каждый раз, когда вы нажимаете <Enter>, хранящиеся в буфере символы обрабатываются, и копия строки выводится. Это продолжается вплоть до эмуляции конца файла в стиле Unix. В другой системе пришлось бы нажать комбинацию <Ctrl+Z>.

Давайте подумаем о возможностях, доступных для программы echo_eof.c. Она копирует на экран любые переданный ей ввод. Предположим, что вы каким-то образом предоставили ей файл. Тогда программа выведет на экран содержимое этого файла, остановившись по достижении конца файла после обнаружения сигнала EOF. А еще представим, что вместо этого вы нашли способ направить вывод программы в файл. Тогда можно было ввести данные с клавиатуры и использовать echo_eof.c для их сохранения в файле. Далее предположим, что вам удалось сделать то и другое одновременно: направить ввод из одного файла в echo_eof.c и переслать вывод в дру-

гой файл. Тогда программу echo_eof.c можно было бы применять для копирования файлов. Эта небольшая программа обладает потенциалом для просмотра содержимого файлов, создания новых файлов и копирования существующих файлов — весьма неплохо для такой короткой программы! Ключом является управление потоком ввода и вывода, что и будет следующей рассматриваемой темой.

НА ЗАМЕТКУ! Эмулированный символ EOF и графические интерфейсы

Концепция эмуляции символа EOF возникла в среде командной строки, использующей текстовый интерфейс. В такой среде пользователь взаимодействует с программой через нажатия клавиш, и сигнал EOF генерирует операционная система. Некоторые действия не очень хорошо транслируются в графические среды, такие как Windows и Macintosh, с более сложными пользовательскими интерфейсами, которые включают перемещение курсора мыши и щелчки на кнопках. Поведение программы, сталкивающейся с эмулированным символом EOF, зависит от компилятора и типа проекта. Например, в зависимости от настроек нажатие <Ctrl+Z> может завершить ввод данных или же завершить выполнение самой программы.

Перенаправление и файлы

С вводом и выводом связаны функции, данные и устройства. Рассмотрим для примера программу echo_eof.c. В ней используется функция getchar(). Входным устройством (согласно нашему предположению) является клавиатура, а поток входных данных состоит из отдельных символов. Представим, что вы хотите сохранить ту же самую функцию ввода и ту же разновидность данных, но изменить место, где программа ожидает найти данные. При этом возникает вопрос: а как программа узнает, откуда получать ввод?

По умолчанию программа на C, в которой применяется стандартный пакет ввода-вывода, считает источником входных данных стандартный ввод. Это поток, ранее идентифицированный как stdin. Он представляет собой все, что было настроено в качестве обычного метода чтения данных в компьютер. Им могут быть такие устаревшие устройства, как магнитная лента, перфокарты или телетайп, либо (что мы будем подразумевать в дальнейшем) клавиатура или какая-то передовая технология наподобие голосового ввода. Однако в современных компьютерах этот поток является настраиваемым инструментом, который можно ориентировать на какое-то другое место. В частности, программе можно указать на необходимость получения ввода из файла, а не с клавиатуры.

Существуют два способа заставить программу работать с файлами. Один из них предполагает явное использование специальных функций, которые открывают, закрывают, читают, записывают в файлы и т.д. Исследование этого метода мы отложим до главы 13. Второй способ заключается в применении программы, спроектированной для работы с клавиатурой и экраном, но перенаправлении ввода и вывода в разные каналы — например, в файл и из файла. Другими словами поток stdin переназначается в файл. Функция getchar() продолжает получать данные из потока, в действительности не интересуясь, откуда поток получает свои данные. Такой подход (перенаправление) в некоторых аспектах является более ограниченным, чем первый подход, но он намного проще в использовании и позволяет ознакомиться с распространенными приемами обработки файлов.

Одна из главных проблем перенаправления состоит в том, что оно связано с операционной системой, а не с языком C. Однако многие среды с языком C, включая Unix, Linux и режим командной строки Windows, поддерживают перенаправление, а некоторые реализации C эмулируют его в системах, где перенаправление отсутствует. Операционная система Apple OS X действует поверх Unix, и режим командной строки

Unix можно инициировать, запустив приложение Terminal. Мы взглянем на версии перенаправления в Unix, Linux и Windows.

Перенаправление в Unix, Linux и командной строке Windows

Операционные системы Unix (в режиме командной строки), Linux (аналогично) и режим командной строки Windows (который имитирует старую среду командной строки DOS) позволяют перенаправлять как ввод, так и вывод. Перенаправление ввода предоставляет программе возможность применять для ввода файл вместо клавиатуры, а перенаправление вывода — использовать для вывода файл вместо экрана.

Перенаправление ввода

Предположим, что вы скомпилировали программу `echo_eof.c` и поместили ее исполняемую версию в файл по имени `echo_eof` (или `echo_eof.exe` в системе Windows). Чтобы запустить программу, введите имя файла:

```
echo_eof
```

Программа выполняется так, как было описано ранее, получая ввод с клавиатуры. Теперь предположим, что вы хотите применить эту программу к текстовому файлу с именем `words`. *Текстовый файл* содержит текст, т.е. данные хранятся в виде символов, воспринимаемых человеком. Например, это может быть очерк или программа на языке C. Файл, содержащий инструкции машинного языка, такой как файл с исполняемой версией программы, не является текстовым. Поскольку программа работает с символами, она должна использоваться с текстовыми файлами. Все, что понадобится — ввести следующую команду:

```
echo_eof < words
```

Символ < представляет собой операцию перенаправления в Unix, Linux и DOS/Windows. Она приводит к тому, что файл `words` ассоциируется с потоком `stdin` с передачей по каналу содержимого файла в программу `echo_eof`. Сама программа `echo_eof` даже не знает (и не беспокоится об этом), что ввод поступает из файла, а не с клавиатуры. Ей известен только тот факт, что ей поставляется поток символов, поэтому программа читает и выводит по одному символу за раз, пока не будет достигнут конец файла. Поскольку в C файлы и устройства ввода-вывода приравнены друг к другу, файл теперь является *устройством* ввода-вывода. Опробуйте это!

> **НА ЗАМЕТКУ! Дополнительные пояснения по перенаправлению**
>
> В Unix, Linux и командной строке Windows пробелы с обеих сторон знака < не обязательны. Некоторые системы вроде AmigaDOS (упоминается специально для тех, кто ностальгирует за старыми добрыми временами) поддерживают перенаправление, но не разрешают указывать пробел между символом перенаправления и именем файла.

Ниже приводится пример запуска программы `echo_eof` с конкретным текстовым файлом `words`; знак `$` — одно из стандартных приглашений на ввод в Unix и Linux. В Windows/DOS приглашение на ввод может выглядеть как `A>` или `C>`.

```
$ echo_eof < words
Пешеходов надо любить.
Пешеходы составляют большую часть человечества.
Мало того — лучшую его часть.
Пешеходы создали мир.
$
```

Итак, перейдем к сути дела.

Перенаправление вывода

Теперь предположим, что вы хотите, чтобы программа echo_eof пересылала клавиатурный ввод в файл по имени mywords. В этом случае потребуется ввести следующую команду и начать набор:

```
echo_eof > mywords
```

Знак > представляет еще одну операцию перенаправления. Он приводит к созданию нового файла с именем mywords и переадресует в него вывод echo_eof (т.е. копию символов, набираемых на клавиатуре). Перенаправление переназначает stdout с устройства отображения (экрана) на файл mywords. Если файл mywords уже существует, обычно он очищается и заменяется новым содержимым. (Однако многие операционные системы предоставляют возможность защиты существующих файлов, делая их файлами только для чтения.) Все, что вы видите на экране — это символы, набираемые на клавиатуре, а их копии поступают в файл. Чтобы завершить программу, нажмите комбинацию клавиш <Ctrl+D> (Unix) или <Ctrl+Z> (DOS) в начале строки. Попробуйте это сами. Если не можете придумать, что вводить с клавиатуры, просто повторите приведенный ниже пример. В нем присутствует приглашение $ системы Unix. Не забывайте завершать каждую строку нажатием <Enter>, чтобы содержимое буфера отправлялось в программу.

```
$ echo_eof > mywords
У вас не должно возникать никаких проблем с запоминанием того, что делает
тот или иной оператор перенаправления. Запомните только, что оператор
указывает направление потока информации. Представьте себе, что
это воронка.
[Ctrl+D]
$
```

После того, как комбинация <Ctrl+D> или <Ctrl+Z> будет обработана, программа завершится и на экране снова отобразится приглашение на ввод. Выполнила ли программа свою работу? Команда ls системы Unix или команда dir командной строки Windows, которые выводят на экран список имен файлов, должны подтвердить существование файла mywords. Вы можете воспользоваться командой cat в Unix и Linux или type в DOS для проверки его содержимого либо запустить программу echo_eof снова, на этот раз перенаправив файл mywords в программу:

```
$  echo_eof < mywords
У вас не должно возникать никаких проблем с запоминанием того, что делает
тот или иной оператор перенаправления. Запомните только, что оператор
указывает направление потока информации. Представьте себе, что
это воронка.
$
```

Комбинированное перенаправление

Теперь предположим, что вы хотите создать копию файла mywords и назначить ей имя savewords. Достаточно ввести следующую команду:

```
echo_eof < mywords > savewords
```

и дело сделано. Показанная далее команда также сработает, потому что порядок указания операций перенаправления не играет роли:

```
echo_eof > savewords < mywords
```

Однако будьте внимательны: не применяйте один и тот же файл для ввода и вывода внутри одной команды:

```
echo_eof < mywords > mywords   ← НЕПРАВИЛЬНО!
```

Причина в том, что конструкция > `mywords` приводит к усечению исходного файла `mywords` до нулевой длины до того, как он будет использован в качестве ввода.

В двух словах, существуют правила, регламентирующие применение операций перенаправления (< и >) в средах Unix, Linux и Windows/DOS.

- Операция перенаправления соединяет *исполняемую* программу (равно как и стандартные команды операционной системы) с файлом данных. Она не может использоваться для соединения одного файла данных с другим, а также для соединения одной программы с другой.
- С помощью этих операций ввод нельзя получать из более чем одного файла, а вывод направлять в более чем один файл.
- Обычно пробелы между именами и операциями являются необязательными за редким исключением, когда применяются специальные символы, имеющие особый смысл в командной оболочке Unix или Linux либо в режиме командной строки Windows. Например, можно было бы иметь команду `echo_eof<words`.

Вы уже видели несколько правильных примеров. Ниже перечислен ряд некорректных примеров, в которых `addup` и `count` выступают как исполняемые программы, а `fish` и `beets` — как текстовые файлы:

```
fish > beets            ← Нарушает первое правило
addup < count           ← Нарушает первое правило
addup < fish < beets    ← Нарушает второе правило
count > beets fish      ← Нарушает второе правило
```

В средах Unix, Linux и Windows/DOS также доступна операция >>, которая позволяет добавлять данные в конец существующего файла, и операция конвейера (|), делающая возможным соединение вывода одной программы с вводом другой программы. За дополнительными сведениями по всем этим операциям обращайтесь к соответствующим книгам.

Комментарии

Перенаправление позволяет использовать программы, предназначенные для обработки ввода с клавиатуры, с файлами. Для этого в программе должна предприниматься проверка на предмет достижения конца файла. Например, в главе 7 была представлена программа, которая подсчитывала слова до появления первого символа `|`. Измените тип `ch` с `char` на `int` и замените `'|'` на `EOF` в выражении проверки цикла, после чего эту программу можно будет применять для подсчета слов в текстовых файлах.

Перенаправление — это концепция командной строки, т.к. оно задается путем ввода с клавиатуры специальных символов в командной строке. Если вы не используете среду командной строки, возможность применения этого приема по-прежнему доступна. Во-первых, в некоторых интегрированных средах имеются пункты меню, позволяющие указывать перенаправление. Во-вторых, в системах Windows можно открыть окно командной строки и запустить исполняемый файл в командной строке. По умолчанию среда Microsoft Visual Studio помещает исполняемый файл в подпапку `Debug` внутри папки проекта. Имя файла будет иметь то же базовое имя, что у проекта, и расширение `.exe`. По умолчанию система XCode также называет исполняемый файл по имени проекта и помещает его в папку `Debug`. Исполняемый файл можно запустить из

утилиты Terminal, которая запускает соответствующую версию Unix. Однако в случае использования этой утилиты вероятно проще применять один из компиляторов командной строки (GCC или Clang), который можно загрузить из веб-сайта Apple.

Если перенаправление не работает, можете попытаться заставить программу открыть файл напрямую. В листинге 8.3 показан пример с минимальными пояснениями. Более подробную информацию можно найти в главе 13. Предназначенный для чтения файл должен находиться в том же каталоге, что и исполняемый файл.

Листинг 8.3. Программа `file_eof.c`

```
// file_eof.c -- открывает файл и отображает его содержимое
#include <stdio.h>
#include <stdlib.h>          // для функции exit()
int main()
{
   int ch;
   FILE * fp;
   char fname[50];           // для хранения имени файла

   printf("Введите имя файла: ");
   scanf("%s", fname);
   fp = fopen(fname, "r");   // открыть файл для чтения
   if (fp == NULL)           // попытка завершилась неудачей
   {
      printf("Не удается открыть файл. Программа завершена.\n");
      exit(1);               // выйти из программы
   }
   // функция getc(fp) получает символ из открытого файла
   while ((ch = getc(fp)) != EOF)
      putchar(ch);
   fclose(fp);               // закрыть файл

   return 0;
}
```

Сводка: перенаправление ввода и вывода

В большинстве систем с языком С перенаправление можно использовать либо для всех программ через операционную систему, либо только для программ на С посредством возможностей, предоставляемых компилятором С. В следующих примерах `prog` — это имя исполняемой программы, а `file1` и `file2` — имена файлов.

Перенаправление вывода в файл (>)

```
prog >file1
```

Перенаправление ввода из файла (<)

```
prog <file2
```

Комбинированное перенаправление

```
prog <file2 >file1
prog >file1 <file2
```

В обеих формах `file2` применяется для ввода и `file1` для вывода.

Пробелы

Некоторые системы требуют наличие пробела слева от знака операции перенаправления и отсутствие пробела справа от этого знака. Другие системы (например, Unix) допускают наличие пробелов с обеих сторон либо или их отсутствие.

Создание дружественного пользовательского интерфейса

Большинству из нас приходилось писать программы, пользоваться которыми было не особенно удобно. К счастью, в С имеются инструменты для превращения ввода в более гладкий и приятный процесс. К сожалению, изучение этих инструментов поначалу порождает новые проблемы. Цель настоящего раздела в том, чтобы помочь решить часть проблем, препятствующих созданию более дружественного пользовательского интерфейса, который облегчает ввод интерактивных данных и минимизирует эффект от ошибочного ввода данных.

Работа с буферизированным вводом

Буферизированный ввод часто удобен для пользователя, т.к. он предоставляет возможность редактирования входных данных до отправки их в программу, но для программиста он может стать источником дополнительных забот, когда задействован символьный ввод. Как можно было заметить в ряде приводимых ранее примеров, проблема заключается в том, что буферизированный ввод требует нажатия клавиши <Enter> для передачи введенных данных. Это действие пересылает также символ новой строки, который программа должна обработать. Давайте исследуем эту и другие проблемы на примере программы угадывания чисел. Вы выбираете число, а компьютер пытается его угадать. В программе применяется довольно скучный метод, но мы сосредоточимся на вводе-выводе, а не на алгоритме. В листинге 8.4 приведена начальная версия программы, которая требует дальнейшей доработки.

Листинг 8.4. Программа guess.c

```
/* guess.c -- неэффективное и чреватое ошибками угадывание числа */
#include <stdio.h>
int main(void)
{
   int guess = 1;
   printf("Выберите целое число в интервале от 1 до 100. Я попробую угадать ");
   printf("его.\nНажмите клавишу y, если моя догадка верна и ");
   printf("\nклавишу n в противном случае.\n");
   printf("Вашим числом является %d?\n", guess);
   while (getchar() != 'y')     /* получить ответ, сравнить с y */
      printf("Ладно, тогда это %d?\n", ++guess);
   printf("Я знал, что у меня получится!\n");
   return 0;
}
```

Вот пример выполнения программы:

```
Выберите целое число в интервале от 1 до 100. Я попробую угадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Ладно, тогда это 2?
Ладно, тогда это 3?
n
Ладно, тогда это 4?
Ладно, тогда это 5?
y
Я знал, что у меня получится!
```

Вопреки ожиданиям алгоритма, реализованного в программе, мы выбрали небольшое число. Обратите внимание на то, что после ввода n программа делает два предположения. Дело в том, что программа читает ответ n как отрицание того, что было загадано число 1, и затем считывает символ новой строки как отрицание того факта, что было загадано число 2.

Одно из решений предусматривает использование цикла while для отбрасывания остатка введенной строки, включая символ новой строки. Дополнительное достоинство такого подхода состоит в том, что ответы вроде no или no way будут трактоваться просто как n. Версия в листинге 8.4 интерпретирует no как два ответа. Ниже показан пример цикла, в котором эта проблема устранена:

```
while (getchar() != 'y')  /* получить ответ, сравнить с y */
{
   printf("Ладно, тогда это %d?\n", ++guess);
   while (getchar() != '\n')
      continue;          /* пропустить оставшуюся часть входной строки*/
}
```

В случае применения этого цикла получается следующий вывод:

```
Выберите целое число в интервале от 1 до 100. Я попробую угадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Ладно, тогда это 2?
no
Ладно, тогда это 3?
no sir
Ладно, тогда это 4?
forget it
Ладно, тогда это 5?
y
Я знал, что у меня получится!
```

Проблема с символом новой строки решена. Тем не менее, вряд ли можно посчитать нормальным тот факт, что f трактуется как n. Для устранения этого дефекта можно воспользоваться оператором if, чтобы отфильтровать другие ответы. Прежде всего, определите переменную типа char для хранения ответа:

```
char response;
```

Затем внесите изменения в цикл, чтобы он приобрел следующий вид:

```
while ((response = getchar()) != 'y')   /* получить ответ */
{
   if (response == 'n')
      printf("Ладно, тогда это %d?\n", ++guess);
   else
      printf("Принимаются только варианты y или n.\");
   while (getchar() != '\n')
      continue;             /* пропустить оставшуюся часть входной строки*/
}
```

Теперь вывод выглядит так:

```
Выберите целое число в интервале от 1 до 100. Я попробую угадать его.
Нажмите клавишу y, если моя догадка верна и
клавишу n в противном случае.
Вашим числом является 1?
n
Ладно, тогда это 2?
no
Ладно, тогда это 3?
no sir
Ладно, тогда это 4?
forget it
Принимаются только варианты y или n.
n
Ладно, тогда это 5?
y
Я знал, что у меня получится!
```

При написании интерактивных программ вы должны стараться предвосхищать возможности нарушения инструкций пользователями. Программу необходимо проектировать так, чтобы она элегантно обрабатывала ошибки пользователей. Пользователей следует уведомить о допущенной ошибке и дать дополнительный шанс.

Разумеется, вы должны предоставить пользователю четкие инструкции, однако независимо от того, насколько они ясны, всегда найдутся те, кто интерпретирует их неправильно, а затем обвинит вас в составлении непонятных инструкций.

Смешивание числового и символьного ввода

Предположим, что программа требует символьного ввода с помощью `getchar()` и числового ввода посредством `scanf()`. Каждая из этих функций хорошо делает свою работу, но смешивать их нелегко. Причина в том, что функция `getchar()` читает каждый символ, включая пробелы, символы табуляции и новой строки, в то время как `scanf()` при чтении чисел пропускает пробелы, символы табуляции и новой строки.

Чтобы продемонстрировать проблемы, которые при этом возникают, в листинге 8.5 представлена программа, которая в качестве ввода считывает символ и два числа. Затем она выводит таблицу с этим символом, имеющую столько строк и столбцов, сколько было указано во введенных числах.

Листинг 8.5. Программа showchar1.c

```
/* showchar1.c -- программа с крупной проблемой ввода-вывода */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
   int ch;               /* выводимый символ          */
   int rows, cols;       /* количество строк и столбцов */
   printf("Введите символ и два целых числа:\n");
   while ((ch = getchar()) != '\n')
   {
      scanf("%d %d", &rows, &cols);
      display(ch, rows, cols);
      printf("Введите еще один символ и два целых числа;\n");
      printf("для завершения введите символ новой строки.\n");
   }
   printf("Программа завершена.\n");
   return 0;
}
```

```
void display(char cr, int lines, int width)
{
   int row, col;

   for (row = 1; row <= lines; row++)
   {
      for (col = 1; col <= width; col++)
         putchar(cr);
      putchar('\n');  /* закончить строку и начать новую */
   }
}
```

Обратите внимание на то, что программа читает символ как тип `int`, чтобы сделать возможной проверку на `EOF`. Однако она передает этот символ функции `display()` как тип `char`. Поскольку `char` меньше `int`, некоторые компиляторы предупредят о преобразовании. В данном случае предупреждение можно проигнорировать. Или же вывод предупреждения можно предотвратить, добавив приведение типа:

```
display(char(ch), rows, cols);
```

Программа устроена так, что функция `main()` получает данные, а функция `display()` производит вывод. Давайте взглянем на результаты выполнения программы, чтобы увидеть, в чем заключается проблема:

```
Введите символ и два целых числа:
c 2 3
ccc
ccc
Введите еще один символ и два целых числа;
для завершения введите символ новой строки.
Программа завершена.
```

Сначала программа работает хорошо. Вы вводите `c 2 3`, а программа выводит две строки по три символа `c`, как и ожидалось. Затем она предлагает ввести следующий набор данных и завершает работу, прежде чем вы сможете ответить. Что пошло не так? Проблема снова с символом новой строки, на этот раз с тем, который находится непосредственно после числа `3` в первой введенной строке. Функция `scanf()` оставляет его во входной очереди. В отличие от `scanf()`, функция `getchar()` не пропускает символов новой строки, так что этот символ читается `getchar()` на следующей итерации цикла, прежде чем вы получите возможность ввести что-либо еще. Затем он присваивается переменной `ch`, а равенство `ch` символу новой строки означает завершение цикла.

Чтобы устранить эту проблему, программа должна пропускать любые символы новой строки или пробелы между последним числом, набранным в одном цикле ввода, и первым символом, набираемым в следующей строке. Кроме того, было бы неплохо, если бы в дополнение к проверке `getchar()` программу можно было прекратить на стадии выполнения функции `scanf()`. Все это реализовано в следующей версии программы, показанной в листинге 8.6.

Листинг 8.6. Программа `showchar2.c`

```
/* showchar2.c -- выводит символы в строках и столбцах */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
```

```
    int ch;              /* выводимый символ          */
    int rows, cols;      /* количество строк и столбцов */

    printf("Введите символ и два целых числа:\n");
    while ((ch = getchar()) != '\n')
    {
        if (scanf("%d %d",&rows, &cols) != 2)
            break;
        display(ch, rows, cols);
        while (getchar() != '\n')
        continue;
        printf("Введите еще один символ и два целых числа;\n");
        printf("для завершения введите символ новой строки.\n");
    }
    printf("Программа завершена.\n");
    return 0;
}
void display(char cr, int lines, int width)
{
    int row, col;

    for (row = 1; row <= lines; row++)
    {
        for (col = 1; col <= width; col++)
            putchar(cr);
        putchar('\n'); /* закончить строку и начать новую */
    }
}
```

Оператор `while` заставляет программу пропускать все символы, следующие за вводом `scanf()`, включая символ новой строки. Это подготавливает цикл для чтения первого символа в начале следующей строки. Другими словами, данные можно вводить без ограничений:

```
Введите символ и два целых числа:
c 1 2
cc
Введите еще один символ и два целых числа;
для завершения введите символ новой строки.
! 3 6
!!!!!!
!!!!!!
!!!!!!
Введите еще один символ и два целых числа;
для завершения введите символ новой строки.

Программа завершена.
```

За счет использования оператора `if` вместе с `break` мы завершаем выполнение программы, если значение, возвращаемое функцией `scanf()`, не равно 2. Это происходит, когда одно или оба входных значения не являются целыми числами или встретился символ конца файла.

Проверка допустимости ввода

На практике пользователи программ не всегда следуют инструкциям, и вполне может возникнуть несоответствие между тем, что программа ожидает в качестве ввода, и тем, что в действительности она получает. Такие условия могут привести к аварий-

ному завершению программы. Тем не менее, вероятные ошибки часто можно предугадать и, приложив дополнительные усилия по программированию, заставить программу обнаруживать их и должным образом обрабатывать.

Предположим для примера, что имеется цикл, обрабатывающий неотрицательные числа. Один из видов ошибок, которые может совершить пользователь — ввод отрицательного числа. Для проверки такой ситуации можно предусмотреть выражение отношения:

```
long n;
scanf("%ld", &n);    // получить первое значение
while (n >= 0)       // обнаружить значение, выходящее за пределы диапазона
{
   // обработать n
   scanf("%ld", &n); // получить следующее значение
}
```

Еще одна потенциальная ловушка связана с тем, что пользователь может ввести значение неподходящего типа, такое как символ q. Один из способов обнаружения такого вида ошибок предполагает проверку возвращаемого значения функции scanf(). Как вы помните, она возвращает количество успешно прочитанных элементов; таким образом, выражение

```
scanf("%ld", &n) == 1
```

будет истинным, только если пользователь вводит целое число. Это требует внесения в код следующего изменения:

```
long n;
while (scanf("%ld", &n) == 1 && n >= 0)
{
   // обработать n
}
```

Условие цикла while звучит так: "пока ввод является целочисленным значением и это значение положительно".

В последнем примере цикл прекращается, когда пользователь вводит значение некорректного типа. Однако программу можно сделать более дружественной к пользователю и предоставить ему возможность ввести значение правильного типа. В этом случае понадобится отбросить ввод, который привел scanf() к ошибке при первом вызове, т.к. эта функция оставляет неподходящие данные во входной очереди. Здесь пригодится тот факт, что ввод является потоком символов, поскольку можно воспользоваться функцией getchar() для посимвольного чтения. Все эти идеи можно даже реализовать в виде функции, как показано ниже:

```
long get_long(void)
{
   long input;
   char ch;
   while (scanf("%ld", &input) != 1)
   {
   while ((ch = getchar()) != '\n')
         putchar(ch);  // отбросить неправильный ввод
      printf(" не является целым числом.\nВведите ");
      printf("целое число, такое как 25, -178 или 3: ");
   }
   return input;
}
```

Функция `get_long()` пытается прочитать значение типа `int` в переменную `input`. Если ей это не удастся, происходит вход в тело внешнего цикла `while`. Затем во внутреннем цикле `while` выполняется посимвольное чтение проблемного ввода. Обратите внимание, что функция выбран вариант с отбрасыванием всего, что осталось во входной строке. Другим возможным вариантом может быть отбрасывание следующего символа или слова. Далее функция предлагает пользователю повторить попытку ввода. Внешний цикл продолжает выполняться до тех пор, пока пользователь успешно не введет целое число, что приведет к возврату `scanf()` значения `1`.

После того, как пользователь преодолеет все препятствия, не позволяющие ему вводить целые числа, программа может выяснить, допустимы ли введенные значения. Рассмотрим пример, в котором пользователю требуется ввести верхний и нижний пределы, определяющие диапазон значений. В этом случае в программе наверняка понадобится проверка, не превышает ли первое значение второе (обычно при указании диапазонов предполагается, что первое значение меньше второго). Также может стать необходимой проверка вхождения обоих значений в приемлемые пределы. К примеру, поиск в архиве может не работать со значениями для года, которые меньше 1958 или больше 2014. Такую проверку также имеет смысл реализовать в виде функции.

Рассмотрим одну из возможностей. В следующей функции предполагается, что в код включен заголовочный файл `stdbool.h`. Если в вашей системе тип `_Bool` отсутствует, можете подставить тип `int` для `bool`, `1` для `true` и `0` для `false`. Обратите внимание, что эта функция возвращает `true`, если ввод является недопустимым; отсюда и ее название `bad_limits()`:

```
bool bad_limits(long begin, long end,
                long low, long high)
{
   bool not_good = false;
   if (begin > end)
   {
      printf("%ld не меньше чем %ld.\n", begin, end);
      not_good = true;
   }
   if (begin < low || end < low)
   {
      printf("Значения должны быть равны %d или больше.\n", low);
      not_good = true;
   }
   if (begin > high || end > high)
   {
      printf("Значения должны быть равны %d или меньше.\n", high);
      not_good = true;
   }

   return not_good;
}
```

В листинге 8.7 эти две функции применяются для предоставления целых чисел арифметической функции, которая вычисляет сумму квадратов всех целых чисел в указанном диапазоне. Программа ограничивает верхние и нижние пределы диапазона значениями `1000` и `-1000`, соответственно.

Листинг 8.7. Программа `checking.c`

```
// checking.c -- проверка допустимости ввода
#include <stdio.h>
```

```
#include <stdbool.h>
// проверка, является ли ввод целочисленным
long get_long(void);
// проверка, допустимы ли границы диапазона
bool bad_limits(long begin, long end,
                long low, long high);
// вычисление суммы квадратов целых чисел от a до b
double sum_squares(long a, long b);
int main(void)
{
   const long MIN = -10000000L;     // нижний предел диапазона
   const long MAX = +10000000L;     // верхний предел диапазона
   long start;                      // начало диапазона
   long stop;                       // конец диапазона
   double answer;
   printf("Эта программа вычисляет сумму квадратов "
         "целых чисел в заданном диапазоне.\nНижняя граница не должна "
         "быть меньше -10000000, \na верхняя не должна быть "
         "больше +10000000.\nвведите значения "
         "пределов (для завершения введите 0 для обоих пределов):\n"
         "нижний предел: ");
   start = get_long();
   printf("верхний предел: ");
   stop = get_long();
   while (start !=0 || stop != 0)
   {
      if (bad_limits(start, stop, MIN, MAX))
         printf("Повторите попытку.\n");
      else
      {
         answer = sum_squares(start, stop);
         printf("Сумма квадратов целых чисел ");
         printf("от %ld до %ld равна %g\n",
         start, stop, answer);
      }
      printf("Введите значения пределов (для завершения "
             "введите 0 для обоих пределов):\n");
      printf("нижний предел: ");
      start = get_long();
      printf("верхний предел: ");
      stop = get_long();
   }
   printf("Программа завершена.\n");
   return 0;
}

long get_long(void)
{
   long input;
   char ch;
   while (scanf("%ld", &input) != 1)
   {
      while ((ch = getchar()) != '\n')
         putchar(ch);  // отбросить неправильный ввод
      printf(" не является целочисленным.\nВведите ");
      printf("целое число, такое как 25, -178 или 3: ");
   }
   return input;
}
```

```
double sum_squares(long a, long b)
{
   double total = 0;
   long i;

   for (i = a; i <= b; i++)
      total += (double)i * (double)i;

   return total;
}

bool bad_limits(long begin, long end,
                long low, long high)
{
   bool not_good = false;

   if (begin > end)
   {
      printf("%ld не меньше чем %ld.\n", begin, end);
      not_good = true;
   }
   if (begin < low || end < low)
   {
      printf("Значения должны быть равны %d или больше.\n", low);
      not_good = true;
   }
   if (begin > high || end > high)
   {
      printf("Значения должны быть равны %d или меньше.\n", high);
      not_good = true;
   }

   return not_good;
}
```

Ниже приведены результаты выполнения этой программы:

```
Эта программа вычисляет сумму квадратов целых чисел в заданном диапазоне.
Нижняя граница не должна быть меньше -10000000,
а верхняя не должна быть больше +10000000.
Введите значения пределов (для завершения введите 0 для обоих пределов):
нижний предел: low
low не является целочисленным.
Введите целое число, такое как 25, -178 или 3: 3
верхний предел: a big number
a big number не является целочисленным.
Введите целое число, такое как 25, -178 или 3: 12
Сумма квадратов целых чисел от 3 до 12 равна 645
Введите значения пределов (для завершения введите 0 для обоих пределов):
нижний предел: 80
верхний предел: 10
80 не меньше 10.
Повторите попытку.
Введите значения пределов (для завершения введите 0 для обоих пределов):
нижняя граница: 0
верхняя граница: 0
Программа завершена.
```