Технология CGI

Открываем шлюзы

Web-формы, впервые предусмотренные еще в HTML 2.0, предоставляют пользователю возможность передавать на сервер практически любые данные. Отправленная информация при соответствующих условиях может быть обработана на стороне сервера, в результате чего посетитель web-сайта, нажав «самую главную» кнопку HTML-формы — «Submit», получит уникальный, динамически сгенерированный ответ.

Сегодня Всемирная паутина немыслима без динамических сайтов. Серверных технологий, позволяющих генерировать страницы «на лету» в соответствии с запросами клиента, существует великое множество. Это, например, ASP, PHP, ColdFusion, Java-сервлеты.

Старейшей же серверной технологией, обеспечивающей возможность создания истинно динамических web-сайтов, является CGI (Common Gateway Interface, интерфейс стандартного шлюза). Несмотря на серьезную конкуренцию со стороны перечисленных выше молодых и перспективных механизмов обработки данных на стороне сервера, CGI, обладая прекрасной гибкостью, незатейливой простотой и неприхотливостью по отношению к машинным ресурсам, еще долгое время будет сохранять позиции фаворита разработчиков в своей сфере.

Об особенностях программирования CGI-скриптов уже, казалось бы, давно все сказано. Но далеко не все web-мастеры, ставшие таковыми два-три года назад и привыкшие к роскоши многочис- »

ленных подключаемых модулей Perl и библиотек стандартных скриптов, предоставляемых хостинг-провайдерами, ведают, что когда-то CGI-скриптами вполне могли считаться shell-сценарии и даже пакетные файлы (да-да, те самые, из DOS, с расширением BAT, если в качестве серверной платформы использовалась Windows NT). Самое главное, что все это работало, причем зачастую намного надежнее, чем большинство сегодняшних web-приложений, состоящих из тысяч строк кода. Хотя, бесспорно, такие вещи всесторонне описаны действительно очень давно.

Разумеется, вспоминать об этом сегодня (на уровне практических экспериментов) вряд ли целесообразно. Но и напрочь отметать опыт «старших товарищей» несправедливо. Ведь и в области программирования полноценных Perl-скриптов сокрыта масса не вполне очевидных для новичков нюансов, быть может, даже явным образом нигде так и не рассмотренных. К тому же есть вещи, которые не мешает лишний раз повторить.

И это наши методы?

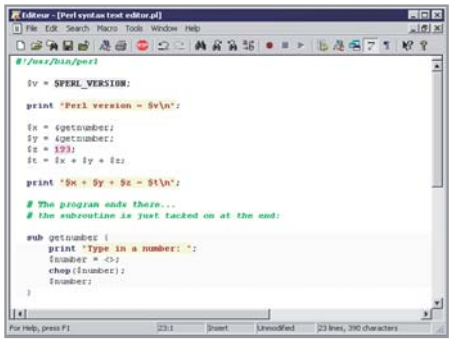
Есть два основных метода передачи пользовательских данных от клиента серверу — POST и GET. А задумывались ли вы над тем, почему у них именно такие названия? Ведь post в буквальном переводе с английского — «отправлять, посылать по почте», а get — «брать, получать».

Порой даже тот, кто в деталях представляет себе, как происходит взаимодействие с сервером в случае передачи клиентом пользовательских параметров различными способами, обычно оказывается не в состоянии объяснить смысл этой терминологической головоломки. Между тем именно знание буквальных переводов слов post и get способно дать новичку возможность никогда не путать соответствующие методы и четко понимать их суть.

Название метода GET подразумевает, что пользователь хочет «взять», «получить» web-страницу, обладающую определенным URL. Например, таким:

```
www.mywebsite.ru/cgi-bin/script.pl?par1=
val1&par2=val2&par3=val3
```

С формальной точки зрения совсем не важно, что после вопросительного



▲ Одна из программ для создания скрипта на Perl называется Editeur

знака следуют какие-то специфичные данные — это просто часть URL.

Метод POST отличается тем, что заданные пользователем значения параметров передаются непосредственно в теле HTTP-запроса, именно отправляются, отсылаются серверу. Причем представляется эта информация точно таким же образом:

```
par1=val1&par2=val2&par3=val3
```

Как видите, никакого противоречия в терминологии нет, названия методов говорят сами за себя.

Но зачем нам два метода, неужели не хватает одного? Дело в том, что нет в мире совершенства, идеального и универсального средства, с успехом справляющегося с любой задачей. У каждого из методов свои индивидуальные особенности характера, приемлемые в одних ситуациях и нежелательные в других. «Темные стороны» метода POST компенсируются достоинствами метода GET, и наоборот. Как это обычно бывает в нашем общем деле, в каждом конкретном случае разработчику нужно принимать одно решение из многих, неизбежно отказываясь от одних возможностей в пользу других, причаливать к тому или иному берегу. Или метаться между двух огней, искать компромиссы — такое тоже не редкость.

Метод GET привлекает разработчиков своей простотой. Все переданные таким способом параметры напрямую доступны CGI-скриптам из переменной окружения QUERY_STRING. Для пользователя тоже есть некоторые плюсы: все параметры на виду, и более-менее искушенный серфер может править их значения прямо в адресной строке браузера, во всей полноте ощущая свободу «неофициальной» навигации.

Главный недостаток метода GET — ограниченная длина строки запроса. Всякий URL так или иначе должен иметь разумную протяженность. От Владивостока до Бреста в любом случае не получится: максимальная длина URL для различных серверов, как правило, 256–1024 символов, к тому же в «секвестре» участвуют и некоторые браузеры.

Метод POST, напротив, не налагает ограничений на размер передаваемых данных, и именно в этом заключается основное его преимущество перед GET. К сожалению, на этом достоинства POST заканчиваются.

Органический недостаток метода POST — невозможность поставить прямую гипертекстовую ссылку на страницу, сгенерированную на основе конкретных значений параметров. Отсюда вытекают все проблемы индексирования подобных динамических отчетов поисковыми системами.

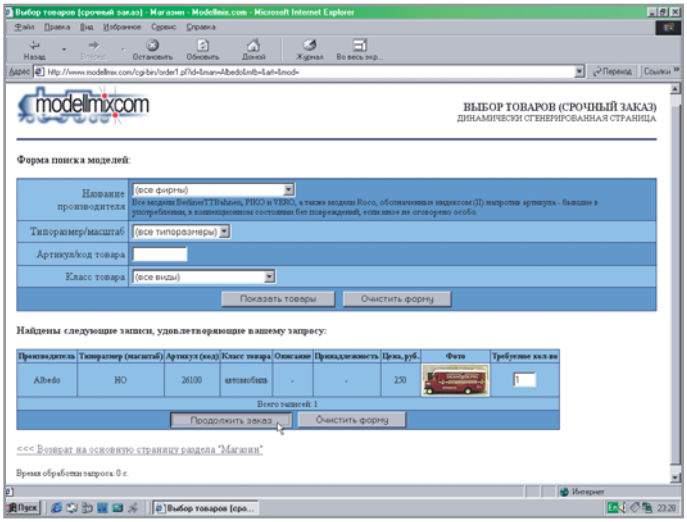
Передача данных

Все и сразу

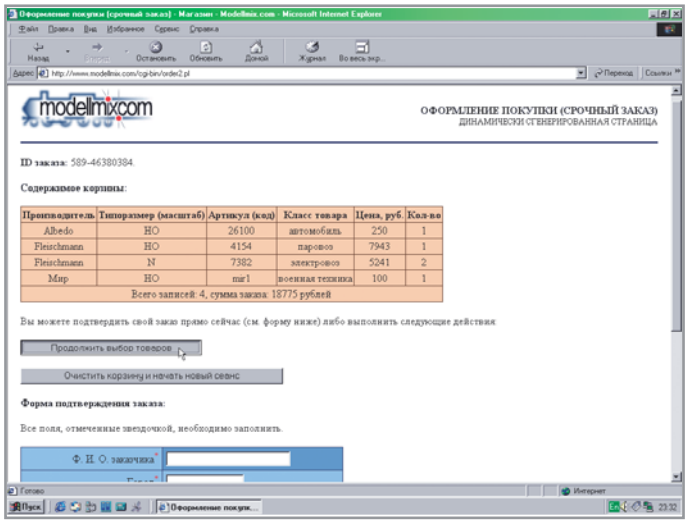
Немногие подозревают, что в рамках единого HTTP-запроса можно передавать параметры серверу одновременно двумя методами. В этом нет ничего сверхъестественного. Двайте рассмотрим пример объявления web-формы:

<form action=«/cgi-bin/script.pl?par1=val1&par2=val2&par3=val3» method=«POST»>

Согласитесь, в атрибуте action мы можем указать любой URL, даже такой, как в примере. Все данные формы будут передаваться серверу, конечно же, методом POST, но строго фиксированные значения параметров par1, par2 и par3 — val1, val2 и val3 соответственно — будут отправлены методом GET. Если дать волю фантазии и сообразительности, можно придумать, как сделать эти значения «не совсем фиксированными». Намекну на некторые: JavaScript, SSI и много чего еще — к вашим услугам!



▲ Данные для формирования этой страницы были переданы методом GET



▲ Эта страница была сгенерирована при помощи метода POST

Еще один существенный недостаток заключается в нарушении логики работы кнопки «Назад» — самого популярного элемента стандартного интерфейса браузера. Вернуться на страницу, сгенерированную при помощи метода POST, после перехода с нее куда-либо еще довольно трудно. Internet Explorer, например, выдаст сообщение «Внимание: страница устарела» с требованием нажать на кнопку «Обновить», если намерения пользователя вновь просмотреть злополучную страницу все-таки серьезны.

В ряде браузеров возникают трудности при сохранении web-страниц, сгенерированных динамически с применением метода POST, для локального просмотра. Отмечались также проблемы с распечаткой подобных документов.

В общем и целом, подводя итог, скажем, что с позиций юзабилити метод POST, мягко говоря, не лучшее решение. Так что при проектировании сложных многошаговых приложений (например, интернет-магазинов) метод POST нужно стараться использовать только там, где требуется передача действительно больших объемов информации. При этом желательно, чтобы логика работы приложения строилась таким образом, чтобы страницы, генерируемые на основе данных, отправляемых методом POST, не требовали возврата к себе.

Специальные формы с кнопками для возвращения к предыдущему шагу на web-страницах организовывать бесполезно — по собственному опыту знаю, что большинство посетителей упрямо потянется к кнопке «Назад». Привычка — вторая натура.

Ручная работа

Почти каждый написанный мной CGI-скрипт начинается так:

```
#!/usr/bin/perl

$query = $ENV{QUERY_STRING};

@query = split('&', $query);
foreach $arg(@query)
{
 @fields = split('=', $arg);
 $fields[0] =~ s/\+/ /g;
 $fields[0] =~ s/%([0-9A-H]{2})/pack('C',
hex($1))/eg;
 $fields[1] =~ s/\+/ /g;
 $fields[1] =~ s/%([0-9A-H]{2})/pack('C',
hex($1))/eg;
 $userdata{$fields[0]} = $fields[1];
}
```

В первой строке как обычно указывается путь к программе — интерпретатору языка Perl. Далее переменной $query присваивается значение переменной окружения QUERY_STRING — это справедливо при использовании метода GET. Если данные HTML-формы пересылаются при помощи метода POST, рассматриваемую строку кода нужно заменить на две:

```
$content_lenght = ENV{CONTENT_LENGTH};
sysread(STDIN, $query, $content_lenght);
```

Затем строка запроса разбивается на части, разделителем выступает символ амперсанда. Из полученных таким образом подстрок формируется массив @query.

Поясним эту ситуацию на примере. Предположим, что сервером принята уже знакомая нам строка запроса: par1 =val1&par2=val2&par3=val3. Элементами массива в этом случае будут: par1 =val1, par2=val2, par3=val3.

Аналогичным образом в цикле для каждого элемента массива @query формируется массив @fields. Его элементами при обработке строки par2=val2 будут являться par2 и val2.

Далее следует ряд регулярных выражений. Если хорошо присмотреться, к каждому из элементов массива @fields применяется пара одинаковых регулярных выражений. В первом происходит замена знака «+» на символ пробела. (Если значения параметров состоят из нескольких слов, обычно при кодировании строки запроса HTTP-клиентом эти части объединяются знаком «+».) Второе регулярное выражение использует функцию pack, для того чтобы преобразовать закодированные последовательности в читабельные символы. Скажем, слово «экскаватор» без соответствующего декодирования выглядело бы так: %FD%EA%F1%EA%E0%E2%E0%F2%EE%F0.

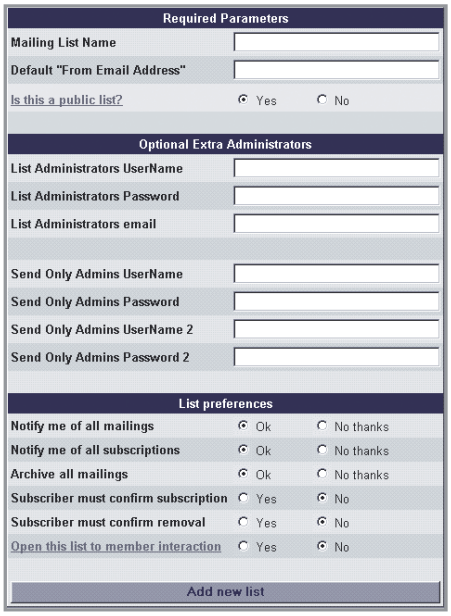
Почему одну и ту же процедуру приходится проделывать для каждого названия и значения всех параметров в отдельности? Нельзя ли сразу же подвергнуть такому преобразованию всю строку запроса в

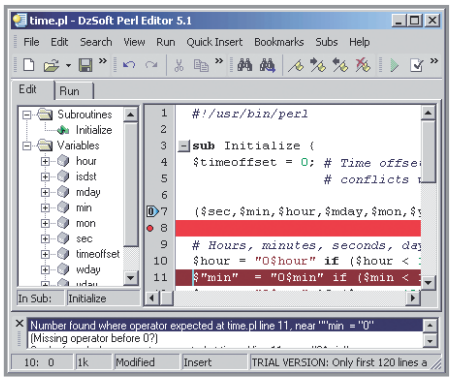
» целом? Дело в том, что мы не имеем права исключать возможность наличия закодированных символов амперсанда («&») и знаков равенства («=») в названиях и значениях параметров. При раннем декодировании этих символов неизбежны ошибки. Посудите сами, ведь тогда амперсанды и знаки равенства, относящиеся к информации пользовательского запроса, станут разделителями, в результате чего полученные названия и значения параметров не будут иметь ничего общего с данными, отправленными клиентом.

Из декодированных элементов массива @fields формируется ассоциативный массив (хэш) %userdata. Это очень удобная структура данных, состоящая из ключей и соответствующих им значений. Как вы понимаете, в качестве ключей полученного хэша выступают названия переданных скрипту параметров, а значения последних ставятся им в соответствие. К примеру, после обработки нашей строки запроса значением скаляра $userdata{'par2'} будет val2.

Собственно, задача разбора строки запроса как раз и заключается в формировании подобного хэша. Мы ее успешно решили.

Недостатком конструкции, приведенной выше, является большое количество глобальных переменных, использующихся для служебных целей только в данном фрагменте программы и нигде более. Эта проблема, конечно же, решаема: необходимо выделить соответствующую часть описанного кода в отдельную функцию и объявить в ней все вспомогательные переменные локальными. Но несложные скрипты вполне можно писать и «сплошным текстом», как в стародавние времена, так получается даже нагляднее.





▲ Еще одна популярная утилита под названием DzSoft Perl Editor, в которой многие опытные разработчики пишут свои скрипты

◀ Пример многоступенчатой CGI-формы, в которой присутствуют распространенные типы ввода информации (radiobutton, textbox)

Вечные испытания

Многие используют ключ -w при указании пути к интерпретатору Perl:

```
#!/usr/bin/perl -w
```

В этом случае интерпретатор Perl выводит разного рода предупреждения (warnings — первая буква именно этого слова, вероятно, и стала поводом для названия ключа) и сообщения об ошибках. При исполнении CGI-скриптов вывод осуществляется, как правило, в файл журнала ошибок (error log) сервера.

Ключ -w вполне можно использовать при первоначальной отладке скрипта, но вряд ли целесообразно применять его »

Преимущества Perl

Легче, свободнее, быстрее!

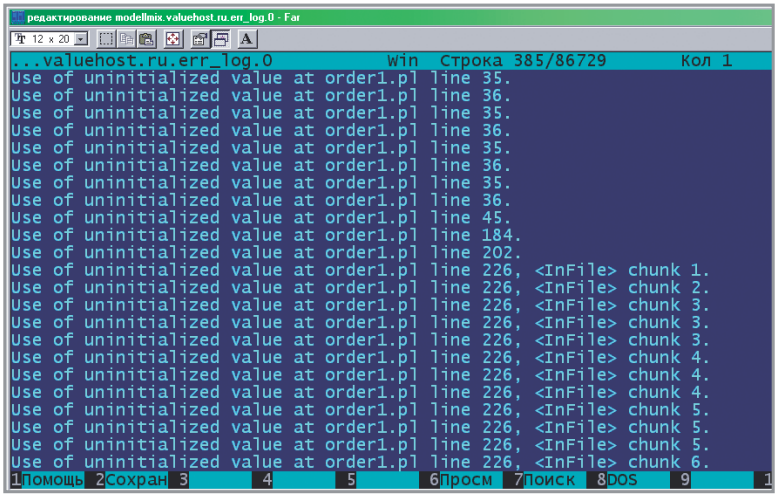
Нынешнюю ситуацию в сфере CGI-программирования можно кратко охарактеризовать, перефразируя известный большевистский лозунг. Мы говорим CGI — подразумеваем Perl. Мы говорим Perl — подразумеваем CGI.pm.

Это притом, что CGI-скрипты можно создавать на любых языках программирования (см. выше о shell-сценариях и BAT-файлах). Для написания критичных в плане быстродействия фрагментов приложений, например, целесообразно использовать C/C++.

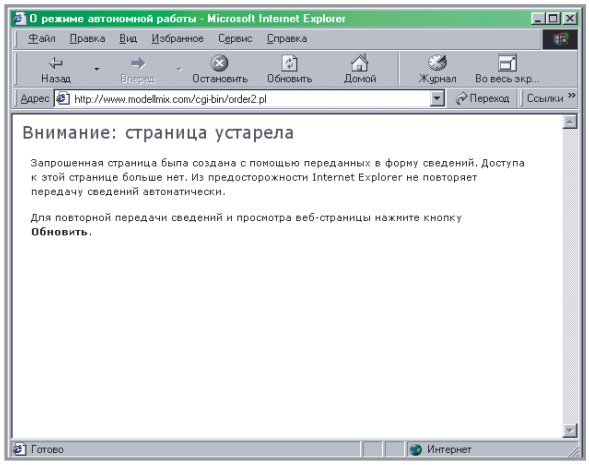
Впрочем, в подавляющем большинстве случаев Perl — действительно лучший выбор. Этот язык прост, понятен и удобен для программиста. Причем настолько, что я не совсем понимаю, какую основу имеет под собой повальное увлечение web-мастеров подключаемым модулем CGI.pm (конструкции которого необходимо дополнительно изучать!) или библиотекой функций cgi-lib.pl (имена и особенности работы каждой из которых нужно, соответственно, тоже запоминать, чтобы не открывать справочник каждые пять минут). Ведь любой CGI-скрипт можно написать с использованием встроенных ресурсов языка Perl, что называется, вручную.

Согласитесь, что при создании скриптов, особенно несложных (а таковыми они бывают достаточно часто — даже крупномасштабные приложения в большинстве своем представляют собой, как правило, наборы связанных воедино CGI-сценариев размером в несколько килобайт каждый), масса возможностей, предоставляемых громоздкими подключаемыми модулями и библиотеками, не используется. Многие, например, применяют объектно-ориентированный интерфейс модуля CGI.pm только для того, чтобы произвести разбор строки запроса. Не смешно.

Кроме того, существенным минусом внешних модулей является недостаточная гибкость. Скажем, в библиотеке cgi-lib.pl имеется функция HtmlBot, которая только и делает, что возвращает строку «</BODY>\n</HTML>\n». Неужели есть люди, которым бывает лень набить эти несчастные 18 байт вручную?! А если, положим, я привык набирать тэги строчными, а не прописными буквами? И вообще хочу перед «</BODY>» вставить еще «</TABLE>\n»?



▲ Если продолжать использовать отладочный ключ -w в штатном режиме работы сложного web-приложения, размер серверного лога ошибок может выйти из-под контроля



▲ Такое бывает при попытке возврата с помощью кнопки «Назад» к странице, сгенерированной на основе информации, переданной методом POST

» в штатной работе. Дело в том, что даже правильно работающий скрипт может при определенных условиях порождать достаточно большое количество предупреждений, которые будут немилосердно засорять серверный лог.

Чтобы не быть голословным, приведу такой пример. Предположим, у нас имеется HTML-форма с теми же самыми тремя полями, использующая метод GET. Если даже не заполнять некоторые поля формы при работе с ней, после нажатия на кнопку «Submit» серверу в любом случае будет передана строка запроса, состоящая из трех параметров. Пусть мы определили значение только второго параметра, а первое и третье поля оставили пустыми. В этом случае URL страницы динамического отчета будет выглядеть так:

```
http://www.mywebsite.ru/cgi-bin/
script.pl?par1=&par2=val2&par3=
```

Но ведь кто-то может поставить ссылку на ту же самую страницу, опустив не только значения, но и названия незадействованных параметров:

```
http://www.mywebsite.ru/cgi-bin/
script.pl?par2=val2
```

Мы не в силах запретить кому-либо ставить подобные ссылки. Более того, они смотрятся элегантнее. Но ведь этого не объяснишь CGI-скрипту, который все равно предполагает, что ему переданы все три параметра HTML-формы. Он по-прежнему будет анализировать значения $userdata{'par1'} и $userdata{'par3'}. Но эти значения в данном контексте будут уже не нулевыми, а неопределенными (undefined, uninitialized), в результате чего в журнал ошибок посыпятся предупреждения, подобные такому:

```
Use of uninitialized value at script.pl
line 35.
```

К слову, масштабы «учебной тревоги» могут быть весьма впечатляющими. Если CGI-скрипт обрабатывает какую-нибудь базу данных в тысячу записей, сравнивая в цикле значения полей каждой записи этой базы с параметрами, полученными из строки запроса, объем гневных предупреждений, отсылаемых в лог, может весьма существенно превышать размер полезной информации — динамической страницы, возвращаемой пользователю. Процессорное время при этом будет расходоваться тоже нерационально.

Тотальный изоляционизм

Предположим, мы проектируем онлайновый справочник по компьютерным магазинам городов России. Данные планируется организовать достаточно просто — в виде текста с разделяемыми полями. Поскольку жителей Новосибирска вряд ли будут интересовать подобные заведения в Нижнем Новгороде, целесообразно каждому городу выделить свой собственный текстовый файл, потому что обработка текста с разделяемыми полями эффективна только при сравнительно небольших объемах информации. Размеры индивидуальных файлов для каждого из городов могут быть вполне приемлемыми, а вот суммарный объем записей по магазинам всей России наверняка окажется весьма существенным. Если бы в техническом задании была задекларирована возможность поиска магазинов без привязки к конкретному городу, данные пришлось бы неизбежно объединять, но простой текст с разделяемыми полями нам бы тогда уже не пригодился. Для хранения сколько-нибудь значительных объемов информации необходимо прибегать к помощи более-менее серьезных СУБД, таких как MySQL.

Итак, пусть у нас имеется информация по пяти городам. Она хранится в следующих файлах:

- Москва — msk.txt;
- Санкт-Петербург — spb.txt;
- Новосибирск — nsk.txt;
- Нижний Новгород — nnov.txt;
- Екатеринбург — yekburg.txt.

Выбор города предполагается осуществлять в выпадающем списке HTML-формы. Мы могли бы передавать CGI-скрипту, обрабатывающему данные, имена файлов напрямую:

```
<select name=«city»>
<option value=«msk» selected>Москва
<option value=«spb»>Санкт-Петербург
<option value=«nsk»>Новосибирск
<option value=«nnov»>Нижний Новгород
<option value=«yekburg»>Екатеринбург
</select>
```

»

» В скрипте беспечный программист написал бы следующее:

```
$filename = $userdata{'city'} . '.txt';
open(InFile, $filename);
```

Кратко, но небезопасно. Даже обязательное добавление расширения «.txt» к имени файла не спасает от потенциальных злоумышленников. Конструкция крайне уязвима — параметр city, содержащий символы конвейера и перенаправления, способен наделать немало бед на сервере, вызывая любые доступные программы со всевозможными аргументами.

Очевидно, следует прибегнуть к фильтрации данных, запретив все потенциально «нехорошие» символы, но и это не гарантирует полной безопасности — никогда не известно, все ли учтено.

В нашем случае, между тем, выгоднее было бы поступить так:

```
if($userdata{'city'} eq 'msk')
{
 $filename = 'msk.txt';
}
elsif($userdata{'city'} eq 'spb')
{
 $filename = 'spb.txt';
}
elsif($userdata{'city'} eq 'nsk')
{
 $filename = 'nsk.txt';
}
elsif($userdata{'city'} eq 'nnov')
{
 $filename = 'nnov.txt';
}
elsif($userdata{'city'} eq 'yekburg')
{
 $filename = 'yekburg.txt';
}
else
{
  print «<p>Передан неизвестный параметр.</p>\n</body>\
n</html>\n»;
 exit(0);
}

open(InFile, $filename);
```

Бесспорно, последний фрагмент визуально менее изящен. К тому же при добавлении нового города придется внести его в этот «список славы». Зато приложение получилось почти пуленепробиваемым. Несмотря на то что современные методы взлома скриптов обходят многие виды защиты, этот простой и аккуратный кусочек кода сломать практически невозможно. В будущем вам придется столкнуться с подобными сложностями, но поверьте, в современных условиях безопасность важнее изящества.

Артемий Ломов