



## Powershell

 **admin**

Циклы необходимы для повторного выполнения одинаковых операций с использованием разных значений. Есть циклы обрабатывающие данные получаемые из конвейера (**ForEach-Object**) и работающие отдельно (**ForEach**). Рассмотрим все возможные циклы Powershell подробнее.

## **ForEach**

Цикл **ForEach** служит для пошаговой переборки значений из коллекции элементов. Обычно при помощи цикла **foreach** перебирают элементы в массиве. **ForEach** является самым простым для понимания и чаще всего используемым циклом в Powershell. ForEach не работает с конвейером для этого есть **ForEach-Object**. Посмотрим на синтаксис **ForEach**.

```
1 foreach ($item in $collection)
2 {script block}
```

Переменная **\$collection** это массив определенный заранее. Переменная **\$item** – это текущий элемент из **\$collection**. По очереди перебираются все элементы из массива **\$collection**. Далее в фигурных скобках обычно вызывают переменную

**\$item** обращаясь к текущему элементу коллекции. Перейдем к примерам. Рассмотрим простейший сценарий по удалению старых журналов сервера IIS.

```
1 $data=((Get-Date).Date).AddDays(-10)
2 $massiv=Get-ChildItem C:\script\ |Where-Object {$_.LastWriteTime -lt
   $data}
3 foreach ($a in ($massiv).Fullname)
4 {
5     Remove-Item $a
6 }
```

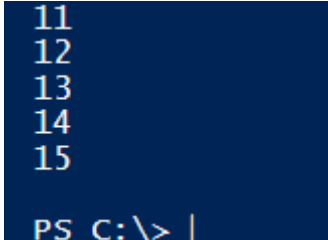
В переменной **\$data** я вычисляю текущую дату минус 10 дней. В **\$massiv** указываю путь к логам `C:\script\` и фильтрую список файлов с датой последней записи позже 10 дней от текущей даты. Далее в цикле **foreach** удаляю каждый файл из переменной **\$massiv**. Скрипт довольно простой но в тоже время полезный.

Рассмотрим пример попроще. Создадим переменную с массивом чисел и в цикле к каждому числу прибавим 10.

```
1 $test=1,2,3,4,5
2 foreach ($name in $test)
3 {
4     $name+10
5 }
```

После прохода цикла **foreach** к каждому числу в массиве **\$test** прибавится 10.

Циклы Powershell **foreach** являются самыми распространенными.



```
11
12
13
14
15
PS C:\> |
```

## ForEach-Object

---

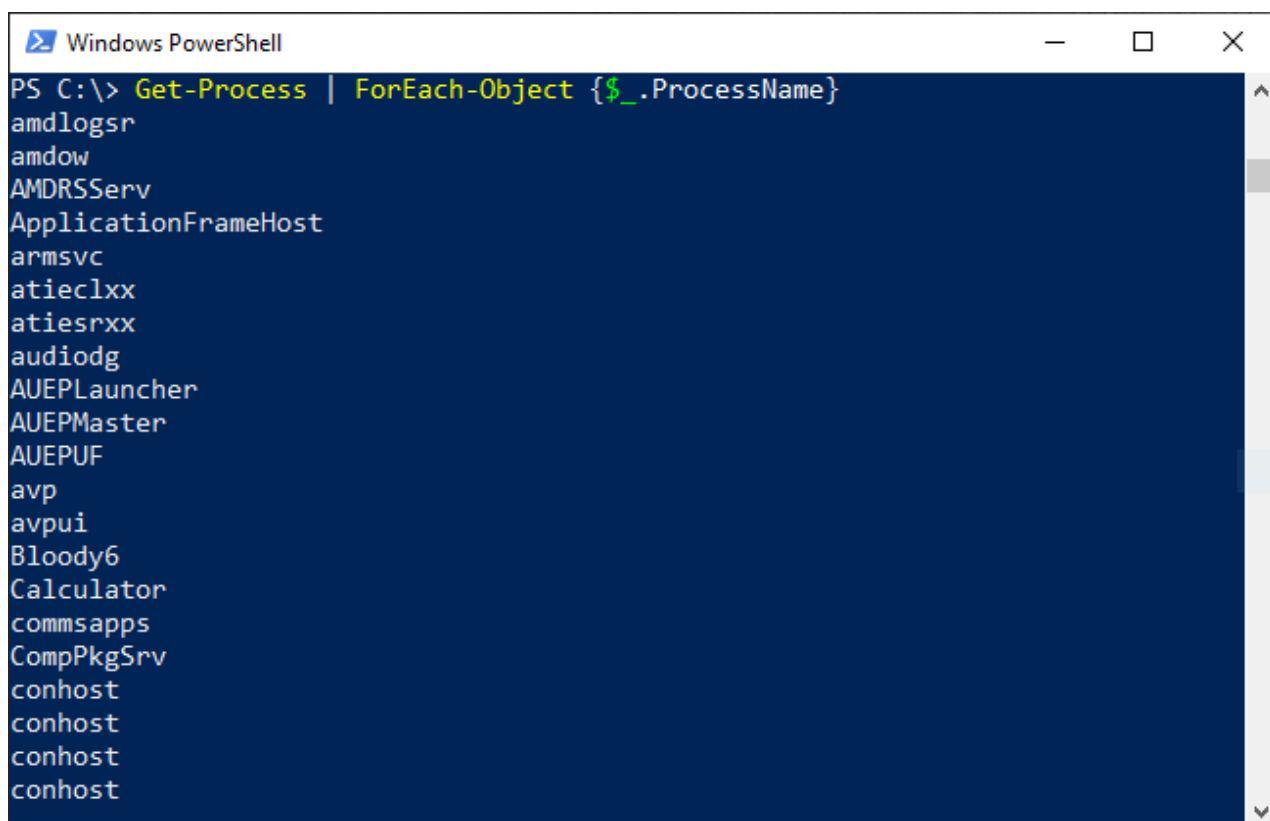
Цикл **ForEach-Object** выполняет операцию над каждым элементом в коллекции входных объектов. Входные объекты передаются командлету **ForEach-Object** по конвейеру или могут быть заданы с помощью параметра **InputObject**. Цикл **ForEach-Object** поддерживает блоки **begin**, **process**, и **end** используемые в функциях.

Существует три способа построения команд в ForEach-Object. Давайте их перечислим.

### Блок сценария

**Блок сценария (Script block)** – для задания операции используется блок скриптов. С помощью переменной `$_` подставляется текущий объект. Блок сценария может содержать любой сценарий PowerShell. Рассмотрим пример с получением списка запущенных процессов.

```
1 Get-Process | ForEach-Object {$_ .ProcessName}
```



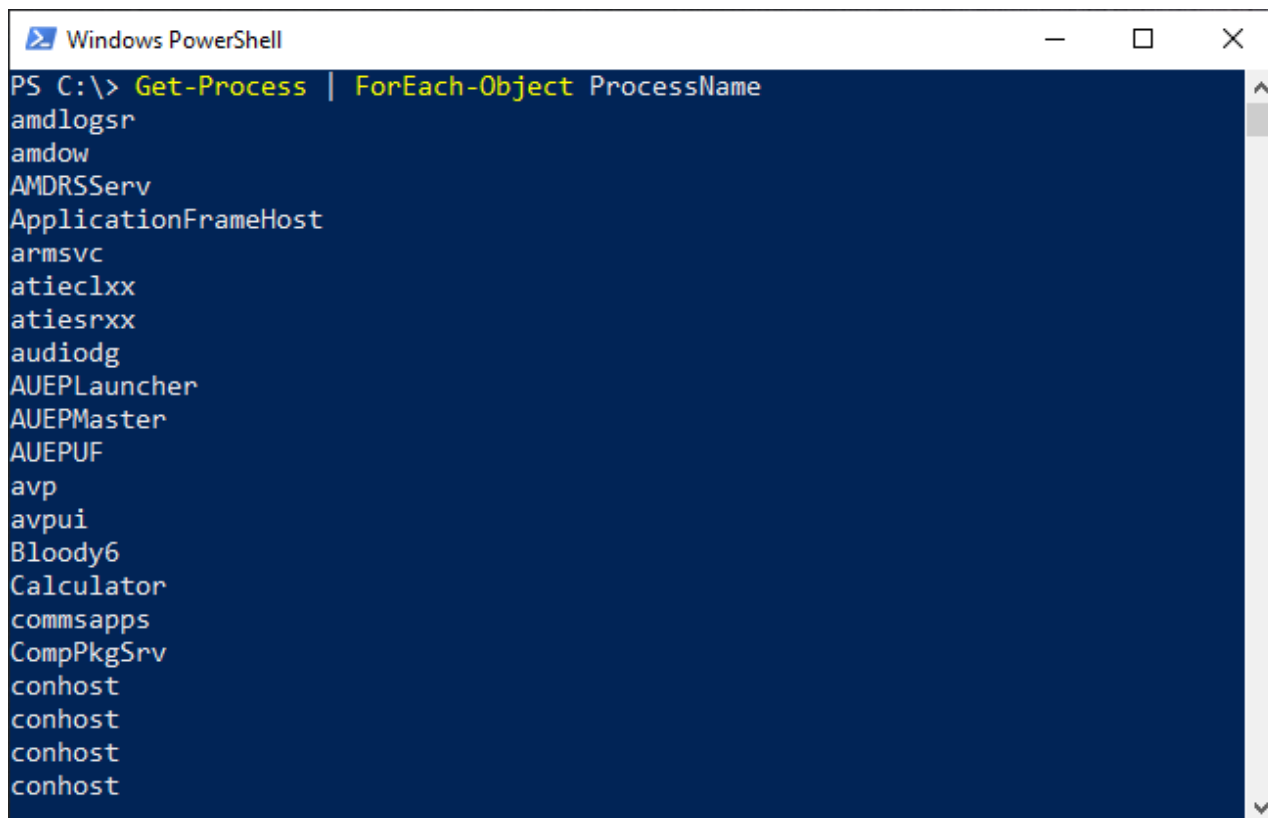
```
Windows PowerShell
PS C:\> Get-Process | ForEach-Object {$_ .ProcessName}
amdlogsr
amdow
AMDRSServ
ApplicationFrameHost
armsvc
atieclxx
atiesrxx
audiodg
AUEPLauncher
AUEPMaster
AUEPUF
avp
avpui
Bloody6
Calculator
commsapps
CompPkgSrv
conhost
conhost
conhost
conhost
```

Блоки сценария выполняются в области действия вызывающего объекта. Т.е. блоки имеют доступ к переменным в этой области и могут создавать новые переменные, которые сохраняются после завершения командлета.

### Оператор выражения

**Оператор выражения (Operation statement)** – в данном случае вы можете использовать оператор сразу указывая значение свойства. Данный способ написания визуально более удобен и проще читается. Эта особенность впервые появилась в Windows PowerShell 3.0. Рассмотрим на примере все того же Get-Process

```
1 Get-Process | ForEach-Object ProcessName
```



```
Windows PowerShell
PS C:\> Get-Process | ForEach-Object ProcessName
amdlogsr
amdow
AMDRSServ
ApplicationFrameHost
armsvc
atieclxx
atiesrxx
audiodg
AUEPLauncher
AUEPMaster
AUEPUF
avp
avpui
Bloody6
Calculator
commsapps
CompPkgSrv
conhost
conhost
conhost
conhost
```

Как видно из примера в данном случае вывод абсолютно такой же как и в случае написания блока сценария.

Сегодня мне поступила задачка, по списку имен сотрудников (список в текстовом файле) вывести соответствующие почтовые адреса из Microsoft Exchange. Делается это одной строкой как раз с использованием ForeEach-Object.

```
1 Get-Content C:\Temp\sotr.txt|ForEach-Object {Get-Mailbox $_ -
  ErrorAction SilentlyContinue}|ft Displayname, WindowsEmailAddress -
  AutoSize
```

На входе у меня список сотрудников в текстовом файле *sotr.txt*. Считанную информацию из файла передаю по конвейеру циклу ForeEach-Object. В цикле командлет Get-Mailbox поочередно для каждого сотрудника из файла считывает информацию и на выходе командлет ft (алиас Format-Table) выводит таблицу с данными: **ФИО – EMAIL**

#### Блок сценария (параллельный запуск)

**Блок сценария (параллельный запуск)** – это новая функция доступна с версии **Windows Powershell 7.0** позволяет запускать блоки сценария параллельно.

Используя параметр **ThrottleLimit** можно ограничить количество одновременно работающих скриптов. В данном случае как и раньше используется переменная **\$\_** для подстановки текущего входного объекта. Используйте **\$using** для передачи ссылок на переменные в запущенный скрипт.

В Microsoft Windows PowerShell 7 для каждой итерации цикла создается новое пространство выполнения, обеспечивающее максимальную изоляцию. В связи с этим нужно четко понимать что объем обрабатываемых данных не займет все ресурсы системы. Если объем данных по циклу большой используйте **ThrottleLimit**. Это позволит ограничить нагрузку на систему и сохранить работоспособность других сервисов. Благодаря параллельному запуску сценарий будет обрабатываться значительно быстрее.

Давайте посмотрим на тестовый скрипт с использованием параллельного запуска.

```
1 Get-ChildItem C:\Windows\ -Recurse|ForEach-Object -Parallel {if
  ($_.Length -ge 100) {$_.Name |Out-File C:\Temp\files.txt -Append}} -
  ThrottleLimit 10
```

Я получаю список всех файлов и передаю их на вход **ForEach-Object**. Цикл параллельно (10 проходов за раз) проходит по каждому файлу и в случае превышения размера более **100 байт** записывает его в файл. Это создает хорошую нагрузку на систему. Давайте посмотрим разницу в загруженных ресурсах.

Последовательно			Параллельно		
Имя	25% ЦП	63% Память	Имя	36% ЦП	71% Память
▼ pwsh (2)	13,9%	118,1 МБ	▼ pwsh (2)	18,6%	1 023,7 МБ
Administrator: PowerShell 7-п...	13,9%	111,5 МБ	Выбрать Administrator: Powe...	18,6%	1 017,1 МБ

Скрипт отработал 1 минуту и за это время при параллельной обработке памяти затрачено **1 ГБ** против **118 МБ** при последовательной. Это практически в **10 раз** больше. Стоит ли скорость отработки затраченным ресурсам решать вам.

## While

Цикл **While** – это языковая конструкция выполняющая команды в командном блоке, пока условие верно. While довольно прост в написании, давайте посмотрим на синтаксис.

```
1 while (<условие>){<выполняемые команды>}
```

Вначале **While** оценивает условие в круглых скобках, если оно верно (**True**) следует выполнение блока команд в фигурных скобках. После первого прохода снова происходит проверка условия и так бесконечно (если его не прервать операторами выхода). Цикл While завершает свою работу если условие становится не верным (**False**).

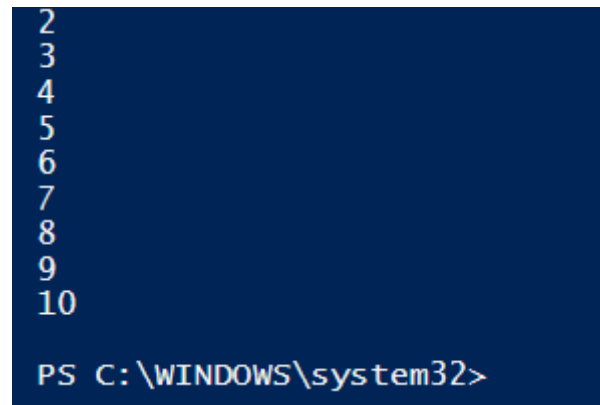
Возьмем простейший пример. Введем новую переменную **\$a** и присвоим ей значение **1**. Далее цикл **while** будет проверять значение **\$a** и пока оно не будет равно **10** выполняется блок команд. В блоке к **\$a** будет прибавляться **1** с каждым проходом цикла. В момент когда **\$a** станет равно 10 цикл остановится.

```
1 $a=1
2 while($a -ne 10)
3 {
4     $a++
5     Write-Host $a
6 }
```

Можно записать цикл и одной строкой, однако читать уже не так удобно

```
1 while($a -ne 10){$a++; Write-Host $a}
```

В данном случае я не задал переменную **\$a** и изначально она пустая. Но с каждой итерацией цикла к ней прибавляется **1**.



```
2
3
4
5
6
7
8
9
10
PS C:\WINDOWS\system32>
```

Еще один пример, пригодится в жизни.

Напишем сценарий постоянно проверяющий запущен ли процесс. Если процесс запущен то ничего не делать, если не запущен то запустить.

```
1 $a=1
2 while($a -eq 1)
3 {
4     if ([bool](Get-Process notepad -ErrorAction SilentlyContinue) -eq $true)
5     {Write-Host "Блокнот запущен! "}
6     else
7     {Start-Process notepad}
8     Start-Sleep -Seconds 30
9 }
```

В данном случае **while** проверяет **\$a=1**, если да то выполнить набор команд. Но **\$a** у меня всегда **1** поэтому цикл будет бесконечный. Это простой пример скрипта для контроля запущенного процесса.

```
Блокнот запущен!  
Блокнот запущен!  
Блокнот запущен!  
Блокнот запущен!  
Блокнот запущен!  
Блокнот запущен!  
Блокнот запущен!
```

```
PS C:\WINDOWS\system32> |
```

## Do

**Do** работает с циклом **While** или **Until** для использования операторов в блоке скрипта в зависимости от условия. Разница между **While** и **Do-While** в том, что блок скрипта в цикле **Do** всегда выполняется как минимум один раз.

В цикле **Do-While** условие вычисляется после выполнения блока скрипта. Так же как в **While** блок скрипта повторяется до тех пор, пока условие оценивается как верное.

```
1 $z = 14  
2 Do {  
3 Write-Host "Z=$z"  
4 $z++  
5 }  
6 While($z -le 20)
```

Цикл **Do-Until** выполняется минимум один раз перед вычислением условия. Однако блок скрипта запускается когда условие ложно. Когда условие станет верным цикл **Do-Until** завершит работу.

```
Z=14  
Z=15  
Z=16  
Z=17  
Z=18  
Z=19  
Z=20
```

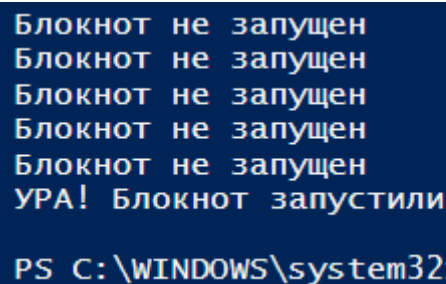
```
PS C:\WINDOWS\system32>
```

```

1 Do
2 {
3   if ([bool](Get-Process notepad -
4     ErrorAction SilentlyContinue)) -eq
5     $False)
6     {Write-Host "Блокнот не запущен"
7       Start-Sleep -Seconds 10}
8   }
9   Until ([bool](Get-Process notepad
10    -ErrorAction SilentlyContinue ))
11
12   Write-Host "УРА! Блокнот запустили"

```

Данный скрипт проверяет запущен ли блокнот. Если не запущен выполняется условие в фигурных скобках после **Do**. Цикл выполняется до тех пор пока блокнот не запустят.



```

Блокнот не запущен
Блокнот не запущен
Блокнот не запущен
Блокнот не запущен
Блокнот не запущен
УРА! Блокнот запустили

PS C:\WINDOWS\system32>

```

## Continue и Break

Операторы **Continue** и **Break** работают со всеми типами циклов, кроме **ForEach-Object**. Они могут работать с метками. Метка – имя которое можно присвоить оператору. Формат задания меток **:metka цикл(условие) {блок скрипта}**.

Оператор **Continue** предоставляет возможность выхода из текущего блока управления. После выхода из блока цикл продолжит выполнение. В момент вызова **Continue** текущая итерация цикла завершается и цикл продолжится со следующей итерацией.

Рассмотрим на примере цикла **While**. Командлет **Get-Process** ищет запущенные процессы **notepad**. Если запущен один процесс **notepad** то сработает **Continue** и цикл продолжится со следующей итерацией. Когда запущено более 1 процесса **notepad** все процессы с данным именем будут закрыты.



```

1 $a=1
2 while($a -eq 1)
3 {
4     if ((Get-Process notepad -ErrorAction Continue).Count -eq 1)
5     {Continue}
6     else
7     {Stop-Process -Name notepad -ErrorAction Continue}
8     Start-Sleep -Seconds 10
9 }

```

Оператор **Break** позволяет выйти из текущего блока управления. Выполнение продолжается на следующем операторе после блока управления. Данный оператор очень удобен если необходимо выйти из постоянно повторяющегося цикла.

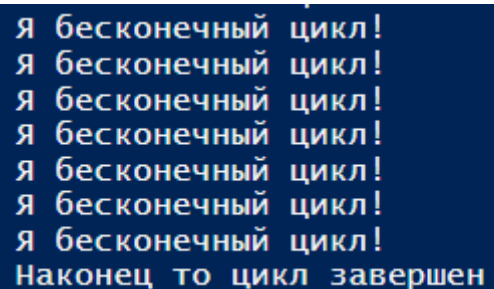
Рассмотрим пример с постоянно повторяющимся циклом, остановить который сможет только запуск процесса ***notepad***

```

1 $z=1
2 while ($z=1) {
3     Write-Host "Я бесконечный цикл!"
4     Start-Sleep 3
5     if (Get-Process -Name notepad -ErrorAction SilentlyContinue) {break}
6 }
7 Write-Host "Наконец то цикл завершен"

```

Операторы Continue и Break отлично дополняют циклы Powershell еще больше расширяя их возможности.



```

Я бесконечный цикл!
Я бесконечный цикл!
Я бесконечный цикл!
Я бесконечный цикл!
Я бесконечный цикл!
Я бесконечный цикл!
Наконец то цикл завершен

```

---

**For**

Цикл **For** – обычно используется для создания цикла, выполняющего команды в командном блоке пока указанное условие оценивается как верное (**\$True**).

Рассмотрим простейший пример цикла **for**

```
1 for ($z=2; $z -le 20; $z++)  
2 {  
3     Write-Host "Текущее значение переменной z="$z  
4 }
```

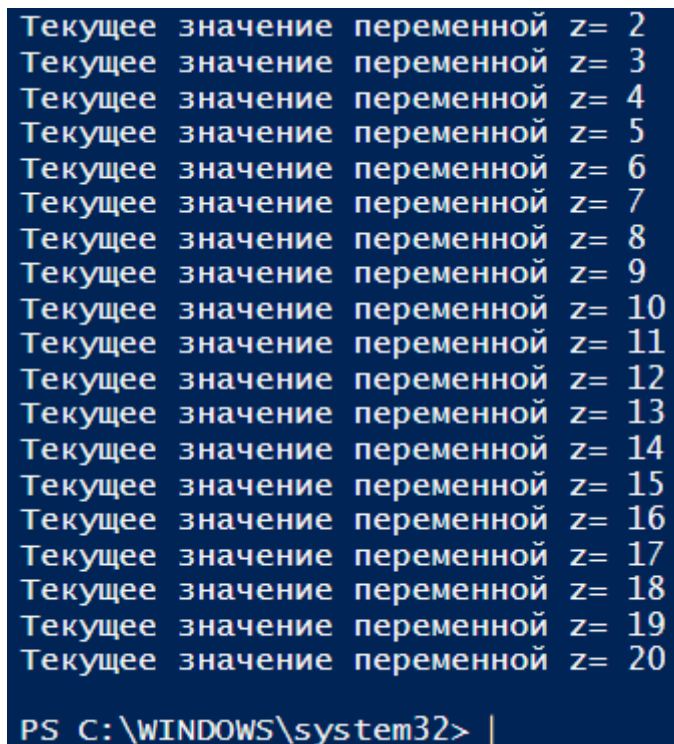
В цикле я задаю переменной **z** значение **2**. Блок команд в фигурных скобках выполняется по **\$z** меньше или равно **20**.

В данной статье я рассмотрел все циклы Powershell существующие на данный момент. Какие из них использовать в работе конечно же решать вам. До новых встреч 😊

П.С.: задать вопросы, пообщаться, обсудить статью циклы Powershell можно у меня в [VK](#).

Рекомендую к прочтению:

- [Powershell скрипты](#)
- [Переменные](#)
- [Операторы сравнения](#)
- [Операторы условий](#)



```
Текущее значение переменной z= 2  
Текущее значение переменной z= 3  
Текущее значение переменной z= 4  
Текущее значение переменной z= 5  
Текущее значение переменной z= 6  
Текущее значение переменной z= 7  
Текущее значение переменной z= 8  
Текущее значение переменной z= 9  
Текущее значение переменной z= 10  
Текущее значение переменной z= 11  
Текущее значение переменной z= 12  
Текущее значение переменной z= 13  
Текущее значение переменной z= 14  
Текущее значение переменной z= 15  
Текущее значение переменной z= 16  
Текущее значение переменной z= 17  
Текущее значение переменной z= 18  
Текущее значение переменной z= 19  
Текущее значение переменной z= 20  
  
PS C:\WINDOWS\system32> |
```