Reclaim gigabytes by deleting unneeded files using a Rexx script

Version 1.0
April 19, 2005

By Howard Fosdick

Takeaway

The trick to reclaiming space is identifying the largest files on your hard disk. The example script in this download automates the retrieval of a list of the largest files on a specified hard drive.

Table of Contents

Copyright ©2005 CNET Networks, Inc. All rights reserved.
For more downloads and a free TechRepublic membership, please visit http://techrepublic.com.com/2001-6240-0.html

More space

Few PC users manage their hard disk space efficiently. And, to tell the truth, many of us server administrators are guilty of the same behavior. Take almost any Windows machine, delete the largest ten or twenty obsolete files, and you'll reclaim anywhere between one and ten gigabytes. That's worth the effort even with today's large disks. What are these files? Take a look at the list below. Typical offenders are product distribution files–*.zip, *.exe, and *.cab files–that are no longer needed. You can archive them to CD or DVD, and then erase the originals to reclaim the disk space. The listing also shows a couple CD images, long ago burned to CDs and now forgotten, wasting 1.4 GB on the disk. On consumer PCs you'll find large audio, image, and video files that you can archive and erase to reclaim gigabytes.

Typical Large Files

```
732942336 c:\knoppix\KNOPPIX_V3.7-2004-12-08-EN.iso
732942336 c:\downloaded_utilities_2\KNOPPIX_V3.7-2004-12-08-EN.iso
269088685 c:\oracle_doc\B14117_01_downloaded.zip
269088685 c:\oracle_doc\B14117_01.zip
175950856 c:\easy_cd_5\easy cd creator 5.02 backup xp.zip
148242516 c:\Documents and Settings\Administrator.NULL\My Documents\Image.nrg
139542090 c:\easy_cd_5\Data.Cab
 76699621 c:\WINDOWS\Driver Cache\i386\driver.cab
 51756917 c:\java\j2sdk-1_4_2_04-windows-i586-p.exe
 48800256 c:\all_ora_1\OID\oid_maint_95forwrk.ppt
 47528448 c:\Documents and Settings\Administrator.NULL\Data\SE v1.4.2_04.msi
 39633168 c:\registry_backups\july_09_2004.reg
 34063765 c:\erwin\AFEDM414sp1Trial-b3907.exe
 31865628 c:\Program Files\Java\Update\Base Images\j2sdk1.4.2-b28\tools.zip
 28740828 c:\attache2\FASTPATH.ZIP
 28740828 c:\attache\FASTPATH.ZIP
```

The trick to reclaiming space is identifying the largest files on your hard disk. But how can you produce a sorted listing like the one above?

It's easy to sort files by size within the Windows Explorer, but the GUI only lets you view one directory at a time. Clearly, automation is required. The Windows directory or *dir* command can list files from all directories on the machine, but even with its plethora of switches, you can't produce a nice listing like that above without some sort of extra processing or "filter." So one solution is a series of four events: a *dir* command, a "filter" to clean up the output, a *sort* command to sort the files by size, and an editor to display the results to the user:

```
dir   |   filter   |    sort    |    wordpad
```

You can write this as a pipe or place it all in a single script. I like the idea of a single script that controls the entire process. It gets the directory listing for all files on the machine, tidies it up, sorts it, and presents it to the user through Window's built-in WordPad editor. All you need to code this is a scripting language that can easily issue and control Windows commands.

Let's look at a simple script that takes this direct approach. In a follow-on article, I'll present a more sophisticated alternative, a script that directly reads through the Windows directory structure to develop its own file list. This latter example is *recursive*—the script calls itself in order to accomplish its work. Each call processes a different sub-directory in the file system. Recursion is a powerful programming technique you can apply to a wide range of problems.

Let's start with the simple approach first.

Copyright ©2005 CNET Networks, Inc. All rights reserved.
To see more downloads and get your free TechRepublic membership, please visit http://techrepublic.com.com/2001-6240-0.html.

A quick script

Any number of programming languages can make short work of this task. I like Rexx, a scripting language that combines *power* with *ease of use.* While these two goals normally conflict, Rexx includes a plethora of design techniques to combine them, including simple syntax, free-formatting, case-insensitivity, structured control and modularity, a tiny core of instructions surrounded by a large function set, easy extensibility, and other many features.

You might recall Rexx from the Windows Resource Kits for Windows 2000 and earlier. What you may not realize is that today there nine free and open source Rexx interpreters that run on every platform, from handhelds to mainframes. All Rexx interpreters meet the international Rexx standard and add extra features for particular purposes—like extensions for Windows or Linux programming, full object-orientation, Java compatibility, extra features for handhelds, UNIX functions, and the like. Rexx is a major scripting language with a strong base in the new Europe. **Table A** lists the free Rexx interpreters, their unique strengths, and where you can download them.

Table A: Free Rexx interpreters

Free Rexx Interpreters	Platforms	Features
Regina	Windows, Linux, Unix, BSD, Mac OS, Symbian handhelds, many other platforms	Professional, rock-solid product that runs across many systems and features detailed documentation.
Reginald	Windows	Standard Rexx with many special extensions for Windows programming.
r4	Windows	Standard Rexx with many special extensions for Windows programming.
BRexx	Linux, Unix, Windows, Windows CE, Mac OS, 16- and 32- bit DOS, Amiga OS, others	Very fast, small-footprint Rexx that runs on many platforms, including natively under Windows CE.
Rexx/imc	Linux, Unix, BSD	Well-proven interpreter for the Unix/Linux/BSD universe.
Rexx for Palm OS	Palm OS	Standard Rexx for the Palm OS.
Open Object Rexx	Linux, Unix, Windows	Fully object-oriented superset of standard Rexx.
roo!	Windows	Fully object-oriented superset of standard Rexx.
NetRexx	Java environments	A "Rexx-like" language for Java environments.
IBM REXX	Mainframes (OS, VM, VSE), iSeries i5OS and OS/400, PC-DOS, OS/2	IBM bundles Rexx with all its operating systems, from mainframes down to PCs.

Let's write the script. The code in **Listing A** is a standard Rexx script that runs under any of the Windows Rexx interpreters (such as Regina and Reginald), and implements a simple approach to identifying large files.

Take a look at the script. You'll notice that Rexx is free format and case-insensitive. Space and capitalize code however you like. Adapt the language to your preferences. Comments appear between the characters /* and */ and may be placed on their own lines or intermingled within the code. I started this script with a *comment block* that defines its purpose.

Listing A

```
/*************************************************************/
/* FIND BIG FILES                                            */
/*                                                           */
/* Displays all files on an Windows machine, from largest    */
/* to smallest, for easy identification of big obsolete files. */
/*                                                           */
```


Copyright ©2005 CNET Networks, Inc. All rights reserved.
To see more downloads and get your free TechRepublic membership, please visit http://techrepublic.com.com/2001-6240-0.html.

```
/*************************************************************/

dir_file  = 'xx_dir_list.txt' /* DIR listing goes to this file */
sort_file = 'xx_sortin.txt'   /* Filtered DIR listing is here  */
out_file  = 'xx_sortout.txt'  /* SORTed DIR listing is here    */

'dir c:\  /-c /s > '  dir_file   /* List all files on machine  */


do while lines(dir_file) > 0     /* Process the full file list */

         line = linein(dir_file)       /* Read a line from file list */


         /* Eliminate blank lines and lines that do not have either  */
         /* DIRECTORY or FILE information in them                    */

         if line = ""                              then iterate
         if pos('<DIR>'               , line) > 0 then iterate
         if pos('File(s)'             , line) > 0 then iterate
         if pos('Total Files Listed'  , line) > 0 then iterate
         if pos('Dir(s)'              , line) > 0 then iterate
         if pos('Volume in drive'     , line) > 0 then iterate
         if pos('Volume Serial Number', line) > 0 then iterate


         /* If the line contains a DIRECTORY statement, process it   */

         if pos('Directory of', line) > 0 then do
                  parse value line with  directory  of  directory_name
                  if directory_name <> 'c:\' then
             directory_name = directory_name || '\'
               end

         /* Else if it's a line with FILE information, process it    */

         else do
                  parse value line  with  date  time  am_pm  size  name
             out_line = format(size,15) directory_name || name
             rc = lineout(sort_file, out_line)
         end

end


/* Close the output file, sort it, and display it to the user */

rc = lineout(sort_file)       /* Close the output file        */

'sort /+1 /r ' sort_file '/o' out_file   /* Sort by file size */

'wordpad' out_file            /* Let user view the big files  */

exit 0
```


Copyright ©2005 CNET Networks, Inc. All rights reserved.
To see more downloads and get your free TechRepublic membership, please visit http://techrepublic.com.com/2001-6240-0.html.

Code analysis

Following the comment block, the first few lines of the program assign file names to the variables *dir_file*, *sort_file*, and *out_file*. The first is the file into which the *dir* command listing goes; the second is where the script writes its cleaned-up or "filtered" file list; and, the last is where the Windows *sort* command writes its sorted output.

```
dir_file  = 'xx_dir_list.txt' /* DIR listing goes to this file */
sort_file = 'xx_sortin.txt'   /* Filtered DIR listing is here  */
out_file  = 'xx_sortout.txt'  /* SORTed DIR listing is here    */
```

The next line in the script shows how the script issues the Windows *dir* command. Rexx interprets a line of code, and whatever it does not understand as part of the Rexx language, it sends to the operating system (or any other specified environment), as a command:

```
'dir c:\  /-c /s > '  dir_file   /* List all files on machine  */
```

I've enclosed the first portion of the *dir* command within single quotes, which prevents Rexx from evaluating the code before sending it to the operating system. In this case this is necessary because otherwise Rexx will try to interpret some of the symbols (like the >) incorrectly. But I did not enclose the reference to the variable *dir_file* in quotes, because I want Rexx to interpret this variable into its value (the file name given it in the initial assignment statement). So after evaluation, Rexx sends a command looking like this to the Windows command line:

```
dir c:\  /-c /s >  xx_dir_list.txt
```

The script issues the *dir* command with the command's */s* flag to provide a full listing of all files in all directories on the drive. The */-c* flag ensures that no commas appear in the file sizes, for easy sorting of the file list by size later on. Of course, you'll get a slightly different file list from the *dir* command depending on which switches you use and your version of Windows. For the purposes of this program, it really doesn't matter, as long as the script identifies the biggest files on the machine. You could also read in the drive letter to process, making this script more flexible, but I just hard-coded it as *c:*\ to simplify the example.

The *dir* command output is sent to the file identified by variable *dir_file*. Next, the script processes all the lines in that file in order to eliminate unneeded output. It accomplishes this by a simple *do while loop*, which uses the built-in function *lines* to determine whether there are lines in the input file to process:

```
do while lines(dir_file) > 0     /* Process the full file list */
```

Once inside the *do while loop*, the *linein* function reads one line of data from the file:

```
line = linein(dir_file)       /* Read a line from file list */
```

Like *lines*, *linein* is a Rexx built-in function. Functions return values that are plopped right into the code where they occur. Rexx features a small instruction set surrounded by a large function library. The language is *extensible* in that there are many, many free function libraries you can plug into your code. Following an initial setup statement, scripts use any external function just like a built-in function. This provides a simple, consistent way to extend the power and functionality of the language without increasing its complexity.

The *dir* command output consists of a label providing a sub-directory name and a list of files in that sub-directory. An example appears is shown in **Listing B**. Our objective is to capture the name of each sub-directory, as well as the file names and sizes. We want to eliminate extraneous lines—for example, those that contain <DIR>**,** the summary lines, and any blank lines. These extraneous lines are indicated in red, while the lines containing the sub-directory names are in blue:

Listing B

```
Directory of c:\project_files\SAP Mapping 2\adding datatypes
03/01/2005  09:01 PM    <DIR>          .
03/01/2005  09:01 PM    <DIR>          ..
10/20/2004  03:52 PM           114688 File Layouts for conversions - MF Vs SAP.xls
               1 File(s)         114688 bytes

Directory of c:\project_files\SAP Mapping 2\already in prod

03/01/2005  09:01 PM    <DIR>          .
03/01/2005  09:01 PM    <DIR>          ..
05/04/2004  05:56 PM           410112 d_MD Conversion and Interface Mapping.xls
04/21/2004  05:00 PM            73216 d_MD Conversion Layouts.xls
07/19/2004  05:09 PM           409600 d_MD Conversion Mapping.xls
               3 File(s)         892928 bytes
```


Copyright ©2005 CNET Networks, Inc. All rights reserved.
To see more downloads and get your free TechRepublic membership, please visit http://techrepublic.com.com/2001-6240-0.html.

First we'll eliminate the lines in the *dir* output that contain extraneous information. These are the lines in red plus the blank lines. The code below employs the *pos* built-in function to identify "junk" lines by their content, while the *iterate* instruction skips to the end of the *do while loop*. So this code identifies and skips junk output without writing it to the output file that will shortly be sent to the Windows *sort* command:

```
/* Eliminate blank lines and lines that do not have either  */
      /* DIRECTORY or FILE information in them                    */

      if line = ""                              then iterate
      if pos('<DIR>'              , line) > 0 then iterate
      if pos('File(s)'            , line) > 0 then iterate
      if pos('Total Files Listed' , line) > 0 then iterate
      if pos('Dir(s)'             , line) > 0 then iterate
      if pos('Volume in drive'    , line) > 0 then iterate
      if pos('Volume Serial Number', line) > 0 then iterate
```

If a line contains label information that identifies a sub-directory, the script needs to parse out the directory name so that it can concatenate it to each file name that appears in that directory. This permits the script to display fully-qualified file names to the user in the final output. These directory label lines are indicated in blue in the above example *dir* output.

This code identifies and parses the blue lines and places the directory name into the variable *directory_name*. It also concatenates a final backslash (\) to the *directory_name*, since in the Windows file naming convention the backslash is used to concatenate a directory name to a file name:

```
/* If the line contains a DIRECTORY statement, process it   */

      if pos('Directory of', line) > 0 then do
               parse value line with  directory  of  directory_name
               if directory_name <> 'c:\' then
             directory_name = directory_name || '\'
            end
```

You'll notice that Rexx features the usual set of structured control instructions: various forms of *do*, *if*, *select* (case), *return*, and *exit*. If multiple instructions must be grouped, use a *do* – *end* pair to denote this. In the above code, several lines follow the *then* branch of the *if* instruction, so the branch is enclosed within a *then do* – *end* grouping. If a single instruction follows a *then*, no *do* – *end* pair is required.

The next code section processes a line that contains file information. These are the black lines in the above example *dir* command output. The *parse* instruction parses the line into its constituent components—the date, time, am_pm code, the file size, and the file name. Rexx is great at parsing and string processing. Like Python, regular expressions are not built into the language but are freely obtainable in any of several free function libraries. In this script I opted to write a few more lines of code and produce a more readable listing, rather than developing a complex regular expression to minimize the length of the script. The choice is yours.

```
      /* Else if it's a line with FILE information, process it    */

      else do
               parse value line  with  date  time  am_pm  size  name
               out_line = format(size,15) directory_name || name
               rc = lineout(sort_file, out_line)
      end
```

The blue line in the above code builds the output line to write to the sort file. It formats the file *size*, and then concatenates it with a single intervening space, to the fully-qualified file name. Rexx allows you to concatenate elements either through the explicit concatenation symbol (||), or simply by listing multiple elements on a line. This line uses the *format* function to format the *size*, and then concatenates this to the *directory_name* and the file name**.**

The red line of code above writes the line to the *sort_file* via the *lineout* built-in function. The return code from the function goes into the variable *rc*. You can easily verify results from functions, subroutines, and operating system commands in Rexx, but I kept this code simple by forgoing error-checking.

After the script has processed all the lines in the *dir_file*, parsed them, and written the relevant ones out to the *sort_file*, the script:

1. Closes the output file
2. Issues the Windows sort command to sort the file listing by file size
3. Invokes WordPad to display the sorted file list to the user

Copyright ©2005 CNET Networks, Inc. All rights reserved.
To see more downloads and get your free TechRepublic membership, please visit http://techrepublic.com.com/2001-6240-0.html.

Here are the lines in the script that accomplish this. It's easy to dynamically build and issue operating system commands and then process their return codes and outputs. Rexx is a "glue language" that stitches together other programs and processes with little effort:

```
rc = lineout(sort_file)       /* Close the output file        */

'sort /+1 /r ' sort_file '/o' out_file   /* Sort by file size */

'wordpad' out_file            /* Let user view the big files  */
```

What next

I've shown one way to quickly write a script to identify large, hard-to-locate files that waste disk space. You could write a shorter, more clever program, but I like the idea of scripting an easy language straight from memory. "Fortune-cookie coding"—short, clever, obtuse scripting—is not the best approach. Simple, readable code means more reliable scripts and higher developer productivity.

There are several other approaches you can take to solve this same programming problem. In a future download, I'll present a program that directly accesses the Windows file system to process its directories and files. What makes the program interesting is that it calls itself every time it encounters a sub-directory it needs to process. So we'll explore an example of a *recursive* program that solves the same problem. Recursion is a powerful technique that can be applied to a wide range of programming problems.

The next script also delves a bit deeper into Rexx. Rexx springs from an entirely different scripting tradition than Perl, Bash, Korn, and many of the other popular scripting languages. Its philosophy of "power through simplicity" makes it a nice complement to languages that rely on complex syntax to provide power. You can become fluent in Rexx in a matter of days, but you won't run out of power as your knowledge grows.

Howard Fosdick has worked with most major scripting languages. His book *Rexx Programmer's Reference* starts with an easy tutorial and then covers everything you'll want to know about Rexx, its interfaces, and tools.

Copyright ©2005 CNET Networks, Inc. All rights reserved.
To see more downloads and get your free TechRepublic membership, please visit http://techrepublic.com.com/2001-6240-0.html.

Additional resources

- How to use Windows shell and the Windows Scripting Host functions (Article)
- Detailed specs for a build-your-own backup network solution (Download)
- The Hard Disk Information Tool (Download)

Version history

Version: 1.0
Published: April 19, 2005

Tell us what you think

TechRepublic downloads are designed to help you get your job done as painlessly and effectively as possible. Because we're continually looking for ways to improve the usefulness of these tools, we need your feedback. Please take a minute to drop us a line and tell us how well this download worked for you and offer your suggestions for improvement.

Thanks!

—The TechRepublic Downloads Team

Copyright ©2005 CNET Networks, Inc. All rights reserved.
To see more downloads and get your free TechRepublic membership, please visit http://techrepublic.com.com/2001-6240-0.html.