

Experiment 1 – File Reverse (Introduction)

Purpose of this task is to make you refresh your knowledge of different programming languages. You will have to write a simple application. If necessary learn to find help in online references and API definitions, such as:

- <http://java.sun.com/reference/api/index.html>
- <http://www.cppreference.com/index.html>
- `$ man <function name>`

Please read the tasks description carefully. If you have questions check the section in question, post a message into your google group or write us¹ an email.

Make yourself familiar with SubVersion beside reading the task description. You will have to use it to submit your solutions. And remember to submit your **correct solution within the deadline!**

Contents

Task.....	1
Implementation Notes.....	2
Example usage.....	2
Working Environment.....	2
Requirements.....	2
Hints.....	3
Dealing with files in Java.....	3
Dealing with Files in C.....	3
Dealing with files in C++.....	4
Examples.....	4
C Example.....	5
Java Example.....	5
Organizational Remarks.....	6
Deadline.....	6
Directory Structure.....	6
Using SubVersion (SVN).....	7
General Principles.....	7
Initial Checkout.....	7
The Basic Work Cycle.....	7
Updating.....	7
Adding, Moving, Deleting or Copying	8
Examining Changes (Status, Diff, Revert)	8
Committing	9
Further Reading.....	9

Task

This task is the base to two following tasks. Write a program that ***copies a file and reverses its content byte by byte***. We request you to implement your application in three different programming languages: C, C++ and Java.

Your C and C++ executable shall be called `rcopy`, so they can be executed with

¹ You find our emails addresses on our website: <http://se.inf.tu-dresden.de>

```
$ rcopy file1 file2
```

In Java we want your file to be named RCopy, so execution can be done by

```
$ java -cp . RCopy file1 file2
```

Write a "reverse comparer" (rdiff) to test your solutions in C. Test your solution using a large random file created with the unix command line tool dd. You can work in the lab or at home.

Implementation Notes

Your implementations have to use a buffer of adequate size. Do not read large files completely into memory, because they may be arbitrary large. Do not read or write files byte-wise, this would be too slow.

Example usage

```
$ cat file1
0123456789
$ rcopy file1 file2
$ cat file2

9876543210$ rdiff file1 file2
$ java -cp . Rcopy file2 file3
$ cat file3
0123456789
$ diff file1 file3
$ _
```

Note that file2 starts with a carriage return, because file1 ends with one. Rdiff has no output if both files are „reversed equal“ to each other and it has to exit with 0. Otherwise it has to exit with 1 and prints an error message to stdout.

Working Environment

For C and C++ your working Environment will be cygwin if you are using MS Windows (<http://www.cygwin.com>) or any Linux distribution or UNIX (Knoppix, Debian, Solaris etc.) with GCC. We suggest that you use Linux. At least one of the following tasks will be solvable under Linux, only.

For Java you will have to use SUN Java SDK (<http://java.sun.com/j2se/index.jsp>) or another Java environment => 1.4 that is installed on your computer. Some later task will require AspectJ that runs best with SUN's Java.

Requirements

Apart from needing basic skills in using the languages C, C++ and Java you will have to be able to create an automatic make process.

This is done with make for C and C++. For Java you will have to use Apache Ant. But for this first task it will not be necessary to change to given build files (e.g.: Makefile, build.xml)

Hints

Familiarize yourself with file-IO API of the programming languages you should use. For more informations about these APIs look into their documentation.

Dealing with files in Java

- **class** `java.io.File`
is an abstract representation of file and directory pathnames.
 - **public** `File(String pathname)` **throws** `NullPointerException` (⇒ If the pathname argument is null)
- **class** `java.io.RandomAccessFile`
A `RandomAccessFile` obtains input bytes from a file in a file system. What files are available depends on the host environment.
 - **public** `RandomAccessFile(File file, String mode)` **throws** `FileNotFoundException`
 - **public** `int read(byte[] b)` **throws** `IOException`
 - **public** `void seek(long pos)` **throws** `IOException`
 - **public** `void close()` **throws** `IOException`
- **class** `java.io.FileOutputStream`
A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`.
 - **public** `FileOutputStream(File file)` **throws** `FileNotFoundException`
 - **public** `void write(byte[] b)` **throws** `IOException`
 - **public** `void close()` **throws** `IOException`

For more details look into the Java API documentation.

Dealing with Files in C

- `FILE *fopen (char *filename, char *mode);`
opens a file and returns file handle
filename: name of file
mode: „r“ read, „w“ write, „r+“ read and write
- `size_t fread (void *p, size_t size, size_t count, FILE *f);`
reads data from file and returns number of actually read bytes
p: memory, where the read data shall be stored (must be available in appropriate size)
size * count: number of bytes to be read
f: file handle of file that provides data to be read
- `size_t fwrite (void *p, size_t size, size_t count, FILE *f);`
writes data to a file
p: address of data, that is to be written into the file
size * count: number of bytes to be written
f: file handle of file, in which shall be written
- `void fclose (FILE *f);`
closes a file
f: file handle of file, that shall be closed
- `int feof (FILE *f);`

returns 0 if end of file is not yet reached otherwise 1

- **int** `fseek` (**FILE** *f, **int** offset, **int** whence);
goes to given position in file f and returns 0 when done without error, otherwise -1
The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence.
- whence: SEEK_SET, SEEK_CUR, SEEK_END
- **void*** `malloc` (size_t size);
allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.
- **void** `free` (**void** *ptr);
deallocates memory given by ptr

Dealing with files in C++

- `fstream(const char *filename, openmode mode);`
 - `ifstream(const char *filename, openmode mode);`
 - `ofstream(const char *filename, openmode mode);`
- `fstream`, `ifstream`, and `ofstream` objects are used to do file I/O.
filename: specifies the file to be opened and associated with the stream
mode: defines how the file is to be opened, according to the ios stream mode flags
- `istream &istream::read(char *buffer, streamsize num);`
reads num bytes from the stream before placing them in buffer
 - `istream &istream::seekg(pos_type position);`
sets the internal get-pointer for the istream to given position
 - `streamsize istream::gcount();`
returns the number of characters read by the last input operation
 - `ostream &ostream::write(const char *buffer, streamsize num);`
writes num bytes from buffer to the current output stream
 - `void istream::close();`
closes the associated file stream
 - `bool istream::is_open();`
returns **true** if associated file stream is open, otherwise **false**
 - `bool istream::bad();`
The `bad()` function returns **true** if a fatal error with the current stream has occurred, **false** otherwise.
 - `bool istream::good();`
The function `good()` returns **true** if no errors have occurred with the current stream, **false** otherwise.

Examples

The two example show how to copy a file *without reversing* it. You may use them as a starting point.

C Example

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #define BUFFER_SIZE 1024
4:
5: int main (int argc, char** argv)
6: {
7:     void *buffer; FILE *f1, *f2; int read;
8:     buffer = malloc (BUFFER_SIZE);
9:     f1 = fopen (argv[1], "r"); f2 = fopen (argv[2], "w");
10:    while (!feof (f1))
11:    {
12:        read = fread (buffer, 1, BUFFER_SIZE, f1);
13:        fwrite (buffer, 1, read, f2);
14:    }
15:    fclose (f1); fclose (f2);
16:    free (buffer);
17:    return 0;
18: }
```

Makefile

```
1: copy: copy.o
2:     gcc copy.o -o copy
3:
4: copy.o: copy.c
5:     gcc -c copy.c
6:
7: clean:
8:     rm copy copy.o
```

copy depends on copy.o, which is dependent on copy.c. clean removes all generated files without any dependencies. Look into the make documentation for more details.

Output

```
%> make
gcc -c copy.c
gcc copy.o -o copy
%> ./copy test.txt test2.txt
copying from test.txt to test2.txt
%> make clean
rm copy copy.o
%> _
```

Java Example

```
1: import java.io.*;
2: public class Copy {
3:     public static void main (String[] args) {
4:         try {
5:             RandomAccessFile inFile = new RandomAccessFile (args[0],
6: "r");
7:             FileOutputStream outStream = new FileOutputStream
8: (args[1]);
9:             byte[] buffer = new byte [1024];
10:            int bytesRead;
11:            do {
12:                bytesRead = inFile.read (buffer);
13:                if (-1 != bytesRead)
14:                    outStream.write (buffer, 0, bytesRead);
15:            } while (-1 != bytesRead);
16:            inFile.close ();
17:            outStream.close ();
18:        } catch (IOException e) {
19:            System.out.println ("IO-Exception while: " + e.toString
20: ());
21:        }
```

Organizational Remarks

Deadline

Solve this week's experiment at home or in the lab.

Please be aware that there is a deadline for this task. You can find the exact date published on our website. If you do not hand in your solution until the specified day we will regard your solution as missing so you will not receive your certificate.

Directory Structure

We require a specific directory structure for your solutions. This is a requirement of our automatic testing system. There is no need to create this directories manually, just check it out from the svn repository.

1_rcopy/		
c/		
	Makefile	- C Makefile for rcopy
	rcopy.c	- C implementation of rcopy
cpp/		
	Makefile	- C++ Makefile for rcopy
	rcopy.cc	- C++ implementation of rcopy
java/		
	Rcopy.java	- Java implementation of rcopy
	build.xml	- Apache Ant build file „make for Java“
rdiff/		
	Makefile	- Makefile for rdiff
	rdiff.c	- C implementation of rdiff

Using SubVersion (SVN)

We will work with a source code repository for this lab. The repository is served by SubVersion (svn), a popular Open-Source version control system. You have to submit your solution using the svn client. Once your changes are committed an automatic test routine will be started.

You got an email from your system with your username and password for your labs repository. If you did not get such a message yet, check your email account for the email address you used for jExam subscription (or your FRZ-account if you did not provided an email address). The URL for your personal directory is:

<https://cate.inf.tu-dresden.de/cate/svn/sftlab20XX/<your matrikel number>>

XX = current year

General Principles

SubVersion works with a (central) repository which keeps the full history of your tasks from which you 'check out' or 'update' a local copy. After making changes to your local copy you 'commit' your changes to the repository automatically creating new versions of the respective files.

You may also work with more than one working copy (e.g. one on your FRZ home directory and one on your private computer at home) We strongly recommend that you keep your working copy even after successfully submitting your solutions until the end of the term.

Initial Checkout

The first thing you need to do is to check out our task so that you have all the files of the repository in your working directory. This is done by:

```
$ mkdir <my-working-directory>
$ cd <my-working-directory>
$ svn checkout <path-to-repository>2 --username username
```

where <path-to-repository> is the path string to the labs repository and <my-working-directory> is the directory where you want to place the files in. The username was send to you via email. You will have to authenticate yourself with the password given to you along your username. Each student is only allowed to checkout and checkin in his/her directory.

You now have all the files and directories of the repository in your chosen working directory and can freely add and edit your files.

The Basic Work Cycle

Updating

Before you do any changes make sure your local files represent the current state of the repository. In your working directory (the one that contains the ".svn" directory) type:

```
$ svn update
```

You then get some lines preceded with a letter and followed by a filename. These have the following meanings:

- A** - (added) The (local) file was newly created.
- U** - (updated) The (local) file was changed with changes from the server.
- G** - (merged) The local file was changed and it also was changed on the repository, but both changes were in different parts of the file, so the new changes from the server were applied to the local file.
- C** - (conflict) The local changes are in conflict with the changes to the file in the repository.
- D** - (deleted) the (local) file was deleted
- R** - (replaced)

Adding, Moving, Deleting or Copying

You can freely modify your local files with any editor you like, only the follwing things need special attention:

When you create a new file in your working copy note that it is **not** automatically managed by svn. You have to explicitly add it to the repository by using:

```
$ svn add myfile
```

If you want to delete a file, do **not** simply delete it from you local copy. It will then automatically be re-added to your working copy the next time you update. Use instead:

```
$ svn delete myfile
```

² Labs repository URL: <https://cate.inf.tu-dresden.de/cate/svn/sftlab20XX/><your matrikel number>
XX = current year

The same rules apply for copying or moving files or directories. Type:

```
$ svn copy source-file destination-file
```

and

```
$ svn move source-file destination-file
```

Examining Changes (Status, Diff, Revert)

Now that you have made your changes you can add them to the repository. But first, always make sure your code works, do **never** commit non-working code to a repository that you are using together with other developers - it won't make you any friends.

So here are your tools to make sure your changes won't do any harm:

Executing

```
$ svn status
```

in the project directory will show you what changes you made, and would thus be committed. Please take a look at <http://svnbook.red-bean.com/en/1.1/ch03s05.html#svn-ch-3-sect-5.3> for a full description of what the output means and make sure it represents your intentions. The common output is similar to `svn update` but it corresponds to your working copy.

To see what changes you made to the files use

```
$ svn diff
```

or, if you want to see just the changes for a specific file use

```
$ svn diff myfile
```

If you notice changes you don't want to commit or undo changes like adding or moving a file use

```
$ svn revert myfile
```

Committing

Now that you resolved all your conflicts update again and make sure there are no new/further conflicts. Now you can commit your changes:

```
$ svn commit --message "Added new method foo to clas Bar."
```

It's good style to always provide a log message using `-m` or `--message`.

Further Reading

This is the very basic working process with subversion. To get well-written, more detailed descriptions or information about what you can also do with Subversion, see: <http://svnbook.red-bean.com/>