cpif

COLLABORATORS

	TITLE : cpif	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Jordi Fita	February 6, 2018	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
c50d1e3f3d07	2011-08-20	Added the download buttons to cpif.	jfita
5b11cb565540	2011-08-20	Fixed a couple of formatting errors in cpif.	jfita
d8cfb9969649	2011-08-20	Added the cpif tool.	jfita

Contents

cpif is a simple command line tool written in Vala that overwrites a file with the contents of the standard input if and only if the output file and the data read differ.

This tool is very convenient to build literate programming applications with automation software like make. As make uses the file's modification time to know whether to rebuild a target, if a literate program has multiple source code files to tangle within the same documentation, using *cpif* I can avoid rebuilding these files whose content hasn't changed and prevent long recompilation times.

1 The Program

cpif must retrieve the output file to overwrite from the command line parameters, read everything that it can from the standard input and store the data read to a temporary file, so to not use too much memory, and then compare this temporary file and the destination file, if it exists. If the destination and the temporary files have the same content, cpif deletes the temporary file. Otherwise, overwrites the output file with the temporary.

```
<<main function>>=
int main(string[] args)
{
    try {
        <<get the destination file name from the parameters>>
        <<copy stdin to a temporary file>>
        if (files_are_equal(tempFileName, destFileName)) {
            <<remove temporary file>>
        } else {
            <<overwrite destination file with temporary file>>
        }
        return 0;
    } catch (FileError e) {
        stderr.printf("Error: %s", e.message);
    }
    return -1;
}
```

Thus, cpif must check that the user passed the destination file as a command line parameter. Without this parameter, cpif must show its usage and abort the program execution. If there are more than one parameters, these are ignored, as I only store the first parameter as the output file name.

```
<<get the destination file name from the parameters>>=
if (args.length < 2) {
    stderr.printf("Usage: %s destination\n", args[0]);
    return -1;
}
string destFileName = args[1];
```

Besides the destination file, I also need to create a temporary file with the contents from the standard input to compare to the destination file.

```
<<copy stdin to a temporary file>>=
string tempFileName = copy_to_temp_file(stdin);
```

The function that copies the content from the standard input to a temporary file actually doesn't know from where it is copying the data from. This function expects a `FileStream` parameter that uses to read the data and copy to a new temporary file. After the copy is done, this function returns the name of the file where it copied the data to.

```
<<function to create the temporary file>>=
string copy_to_temp_file(GLib.FileStream input) throws FileError
{
    <<create temporary file>>
    <<open temporary file>>
    <<copy input to temporary file>>

    return tempFileName;
}
```

To create the temporary file, Vala offers various functions. The most obvious function to use in this case is `open_tmp` which given a file name template, creates and opens a file. The only problem I have with this function is that it creates the file to the system's temporal folder. This is not a problem per se, but in most installations the temporal system folder and the home folder are in different file systems, which means that to overwrite the destination file I must *copy* the contents from one file to the other instead of just renaming the temporary file. I want to avoid copying to much if possible.

Given that in most cases the destination file is at in the working directory, what I do instead if creating the temporary file there using `mkstemp`. If this function fails, then I fall back to use `open_tmp`. Fortunately, I end up with the same result with both functions: a file name and an open file description. The only *gotcha* is that `mkstemp` **modifies** the input template and thus I can't pass an string literal and I must use a `string` variable. This is also the reason I have to pass again the same template to `open_tmp` if I couldn't use `mkstemp`.

Additionally, I don't check the output of `open_tmp` because if this function fails it throws a `FileError` exception which is exactly what I want.

```
<<create temporary file>>=
string tempFileName = "cpif-XXXXXX";
int file_descriptor = GLib.FileUtils.mkstemp(tempFileName);
if (file_descriptor < 0) {
    file_descriptor = GLib.FileUtils.open_tmp("cpif-XXXXXX", out tempFileName);
}
```

The temporary file is now already open, but I only have a file descriptor. To use this descriptor with Vala, I have to *wrap* this file with a `FileStream` object. As I don't know whether the input is text or binary, I'll assume that everything is binary data.

```
<<open temporary file>>=
GLib.FileStream output = GLib.FileStream.fdopen(file_descriptor, "wb");
if (output == null) {
    throw new FileError.FAILED("Couldn't open temporary file");
}
```

Now it is only a matter to copy the input data to this output file. I also explicitly close the output stream (i.e., I set it to `null`), but this is due a bug from a previous version of Vala that didn't close files when the variable goes out of scope.

```
<<copy input to temporary file>>=
copy_stream(input, output);
output = null;
```

The function that copies from one stream to another is in a separate function because I also need to do this under certain conditions when copying the temporary over the destination file.

In this copy function I can't use `FileStream` member function `read_line` to read from the input because I assume everything is binary and thus could have end of string characters (`\0`). What I need to do is create an array and read the input chunk by chunk and write each chunk to the output.

One thing I need to be very careful is with the parameters to `FileStream` `read` and `write` member functions. According to the documentation, the first parameter is the data array and the second the size (i.e., the number of bytes to read or to write from that array.) Vala, however, doesn't work this way at all. The second parameter is the **number of times** the **whole** array is going to be read or written to. Hence, I want this parameter to be `1` — the default value — almost always.

The problem is that when writing the output, there are times that I don't want to use the whole array. This is the case when there is not enough data from the input to fill up the array. Unable to specify the number of elements from the array to use to write, I am being forced to *slice* the array from the first element up to the last read character.

Again, due to Vala's compiler errors, this slice must be performed in a sentence by itself or the write doesn't work at all. Even though the documentation states that slicing creates a new array, the compiler is smart enough to use pointers instead. I guess Vala uses a copy-on-write policy for slices.

```
<<function to copy a file over another>>=
void copy_stream(GLib.FileStream input, GLib.FileStream output)
{
    uint8[] input_buffer = new uint8[32768];
    size_t bytes_read = 0;
    while (!input.eof() && (bytes_read = input.read(input_buffer)) > 0) {
        uint8[] output_buffer = input_buffer[0:bytes_read];
        output.write(output_buffer);
    }
}
```

With the standard input's data safely stored in a temporary file, it is now time to check if that file and the destination file are equal. Again, I have to assume that both files contain binary data.

```
<<function to compare whether to files are equal>>=
bool files_are_equal(string sourceFileName, string destFileName) throws FileError
{
    <<open destination file>>
    <<open source file>>
    <<compare source and destination files>>
}
```

Both files are equal if every byte that I read from one of them is the same byte I read from the other and I read the end of file at the same time.

```
<<compare source and destination files>>=
int source = 0;
int dest = 0;
do {
    source = sourceFile.getc();
    dest = destFile.getc();
} while (source == dest && source != GLib.FileStream.EOF);

return source == dest;
```

If the destination file does not exist, then, obviously, the two files are different.

```
<<open destination file>>=
GLib.FileStream destFile = GLib.FileStream.open(destFileName, "rb");
if (destFile == null) {
    return false;
}
```

However, if the source file — the temporary file — does not exist, then this is an error that I need to report.

```
<<open source file>>=
GLib.FileStream sourceFile = GLib.FileStream.open(sourceFileName, "rb");
if (sourceFile == null) {
    throw new FileError.NOENT("Couldn't open source file to compare");
}
```

The last thing to do is delete the temporary file if it is equal to the destination.

```
<<remove temporary file>>=
GLib.FileUtils.remove(tempFileName);
```

Otherwise, overwrite the destination file with the contents of the temporary file.

```
<<overwrite destination file with temporary file>>=
move_file(tempFileName, destFileName);
```

```
<<function to move a file>>=
void move_file(string sourceFileName, string destFileName) throws FileError
{
    <<try to move source over destination>>
    if (rename_result < 0) {
        <<copy source to destination>>
        <<remove source file>>
    }
}
```

To move, I first try to rename the source file like the destination file. This is a very lightweight file system operation and is the preferred method to overwrite the destination file.

```
<<try to move source over destination>>=
int rename_result = GLib.FileUtils.rename(sourceFileName, destFileName);
```

If the rename operation that probably means that the source and destination files are in different file systems. In that case, what I need to do is open both files and copy the contents from the source to the destination file. Here I use again the `copy_stream` function defined earlier.

```
<<copy source to destination>>=
GLib.FileStream sourceFile = GLib.FileStream.open(sourceFileName, "rb");
if (sourceFile == null) {
    throw new FileError.FAILED("Couldn't open file '%s'".printf(sourceFileName));
}

GLib.FileStream destFile = GLib.FileStream.open(destFileName, "wb");
if (destFile == null) {
    throw new FileError.FAILED("Couldn't open file '%s'".printf(destFileName));
}

copy_stream(sourceFile, destFile);

destFile = null;
sourceFile = null;
```

Once all the data is copied, I should delete the source file. This isn't actually required, but I don't like to leave behind temporary files.

```
<<remove source file>>=
GLib.FileUtils.remove(sourceFileName);
```

All the previously defined fragments could be written in a single source code as this:

```
<<*>>=
/*
   cpif -- Copy from stdin to a file if the contents are different.
   <<license>>
 */
<<function to copy a file over another>>

<<function to create the temporary file>>

<<function to compare whether to files are equal>>

<<function to move a file>>

<<main function>>
```

2 Building

A small Makefile should be enough to build and link `cpif` from the source document.

The first thing that I needs to do is to extract the Vala source code from the AsciiDoc document using atangle.

```
<<extract vala source code>>=
cpif.vala: cpif.txt
	atangle $< > $@
```

Although it is possible to compile and link the `cpif` executable directly from the Vala source code using the Vala compiler — `valac` — I prefer to avoid the dependency of this compiler when distributing the source code to third parties and thus I generate the C source code from the Vala input.

```
<<generate C source code>>=
cpif.c: cpif.vala
	valac -C -o $@ $<
```

This generated source code is regular C code and thus can be build with the platform's compiler. However, I need to determine that platform's executable suffix (i.e., *.exe* for Microsoft® Windows®) if I want to have the correct Makefile rules. To detect the system on which the executable is being build, I use the `uname -s` command available in both GNU/Linux and in MinGW.

```
<<determine executable suffix>>=
UNAME = $(shell uname -s)
```

Then, I only need to check whether if I can find the *MINGW* string in the output of `uname`. If `findstring` call's result is the empty string, then I assume that I am building in a platform without executable suffix. This works for most Unix environments (e.g., GNU/Linux and Mac OS® X.)

```
<<determine executable suffix>>=
MINGW = $(findstring MINGW, $(UNAME))
ifneq ($(MINGW),)
EXE := .exe
endif
```

With the correct suffix detected, I can now add the correct rule to build the final `cpif` executable from the C source code. Being written in Vala, the executable must be linked to `gobject-2.0` as well.

```
<<build cpif executable>>=
cpif$(EXE): cpif.c
	gcc -o $@ $< `pkg-config --cflags --libs gobject-2.0`
```

It is sometime useful to have rules that removes the executable and all the build artifacts (i.e., the C source code.) Traditionally this rule is named `clean` and removes all the files that the Makefile itself made. I mark this target `PHONY` in case there is a file also named `clean` in the source directory, in which case make would ignore the rule.

```
<<clean build artifacts>>=
.PHONY: clean

clean:
	rm -fr cpif$(EXE) cpif.vala cpif.c
```

Now I have all the required target rule for cpif. As the first rule is the one that make builds by default, I have the executable rule first and then, by order, their dependences until the original document. After all the dependences, I have the `clean` target. The Makefile's structure, then, is the following:

```
<<Makefile>>=
<<determine executable suffix>>
<<build cpif executable>>

<<generate C source code>>
```

```
<<extract vala source code>>

<<clean build artifacts>>
```

3 License

This program is distributed under the terms of the GNU General Public License (GPL) version 2.0 as follows:

```
<<license>>=
Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License version 2.0 as
published by the Free Software Foundation.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA
```