Scripts to launch `Jpp` binaries: the use of quotes for parameter expansion

R.G. Ruiz

November 17, 2026

Abstract

This document shows how to use scripts to launch binary files that require several arguments of the same type (for instance, several file names).

1 Introduction

`Jpp` based source codes are compiled into binary files that often require the user to pass a certain number of arguments. These arguments may be input and output file names, or many other parameters that is not practical to hard-code [1]. Binary files that need input arguments can be launched from the command line by typing the name of the binary file followed by the different arguments. The `JParser` class defined in `Jpp` allows to analyze the command line and to map each of the options to the corresponding variables declared in the source code. Details on how this class works are explained in reference [1].

When the number of user options requierd by a program becomes very large or when a program needs to be launched many times with different values of a certain parameters, it is practical not to launch the code directly from the command line, but to write the corresponding instructions in a shell script. Scripts allow to define variables and to use them as arguments for launching any program. Each shell has its own way of expanding variables, which should be considered at the time of writing a script. In the following sections examples are shown of how to (and how not to) write scripts in `Python` and `bash` to launch a `Jpp` binary file that requires several arguments. In particular, a correct use of the quotes (`"`) for passing multiple arguments of the same type is shown.

2 Scripts to launch a program that gets multiple arguments of the same class

Imagine that you need to launch a binary file called `JProgram`, and that this program requires multiple files `file1, file2, file3...` that are located in `/path/`. Suppose that the code needs these file names to be passed via the `-f` option. `JParser` allows to do this in two different ways:

[1] to hard-code something means to define it inside the source (`.cc`) files. To avoid unnecessary compilations of the same source code, one should not hard-code parameters that may need to be changed.

```
JProgram -f file1 -f file2 -f file3 ...etc
```

or

```
JProgram -f file1 file2 file3 ...etc
```

The first way is more convenient because specifying the `-f` option before each file name helps to prevent user errors.

2.1 Python

In a python script, one could run `JProgram` in the first way by using the `os.system` function as follows:

```
import os
bin = "JProgram"
path = "/path/"
filelist = ["file1" , "file2" , "file3" , ...]
infiles = ""
for filename in filelist:
    infiles +=" -f "+path+filename

os.system(bin + infiles)
```

The argument of the `system` function is a string, which will be expanded by `Python` into mutiple values or words. Each of these values is delimited by each of the white spaces in the string. The `JProgram` code will identify each of these words with one argument. `JParser` will identify the first `-f` word with the option that requires a file name, and will associate the subsequent word (ie `/path/file1`) with a file name. The next argument would be the second `-f`, then `JParser` would associate the word `/path/file2` with that option, and so on.

For running `JProgram` in the second way one would need to do the following:

```
import os
bin = "JProgram"
path = "/path/"
filelist = ["file1" , "file2" , "file3" , ...]
infiles = ""
for filename in filelist:
    infiles +=" "+path+filename

os.system(bin + " -f \"" + str(infiles) + "\"")
```

In this case, one needs to make sure that the list of file names separated by white spaces is expanded as a single value to be associated with the `-f` option. This can be done by putting the list within quotes. Like this, everything inside the quotes will be regarded as a single argument which `JParser` will split into single file names. If the quotes were omitted, only the first file name

would be associated with the `-f` option. In that case the second file name would be treated as a separate argument. `JParser` would treat it as an option starting with the backslash character \ and would produce an error.

2.2 BASH

Multiple examples of scripts that run `Jpp` programs can be found in the different subdirectories of `$JPP_DIR/examples/`. The simplest way to run the previous program in the first way would be

```
JProgram                    \
    -f "/path/file1"        \
    -f "/path/file2"        \
    -f "/path/file3"        \
    ... etc
```

Nevertheless, this is not convenient if the number of files is very large. In such a case, it would be more convenient to launch the program by following the second approach.

```
infiles="/path/file1 /path/file2 /path/file3 ..."

JProgram                    \
    -f "${infiles}"         \
```

The variable `infiles` contains a string of file names separated by white spaces. In this case the `BASH` instruction `${infiles}` expands the whole list as a single value and associates it with the `-f` option. Note that in this case the quotes in `"${infiles}"` are not necessary because `infiles` is already a string. Nevertheless, it is a good practice to use them in case that what follows the `-f` option is not just a simple string. The quotes ensure that whatever comes after the `-f` option will be expanded as a single value regardless of the whitespaces within.

Note also that the following instructions to launch the program following the first approach would not work.

```
infiles="-f /path/file1 -f /path/file2 -f /path/file3 ..."

JProgram                \
    ${infiles}          \
```

Since the full string is expanded as a single value, the second `-f` in the string would be associated to the first `-f` and would be regarded as the name of a file that does not exist and the parser would produce an error.

3 Passing elements of `std` containers

A particular case in which multiple values are passed as command line arguments to the same option, is the case where that option is assoiated to an `std` container such as an `std::vector`. In that case, `JParser` reads and store all the available values in the corresponding `std` container.

For instance, the class `JMultipleFileScanner` is derived from `std::vector<std::string>` and is used to read objects from lists of file names. If a command line option `-f` is associated to a variable `var` of the `JMultipleFileScanner` class and a list of file names is provided between quotes as explained in section 2,

```
-f "file1 file2 file3 ..."
```

`JParser` will read each of the file names between quotesdrigo and store in `var`.

References

[1] M. de Jong. *The C++ Parser Class*.