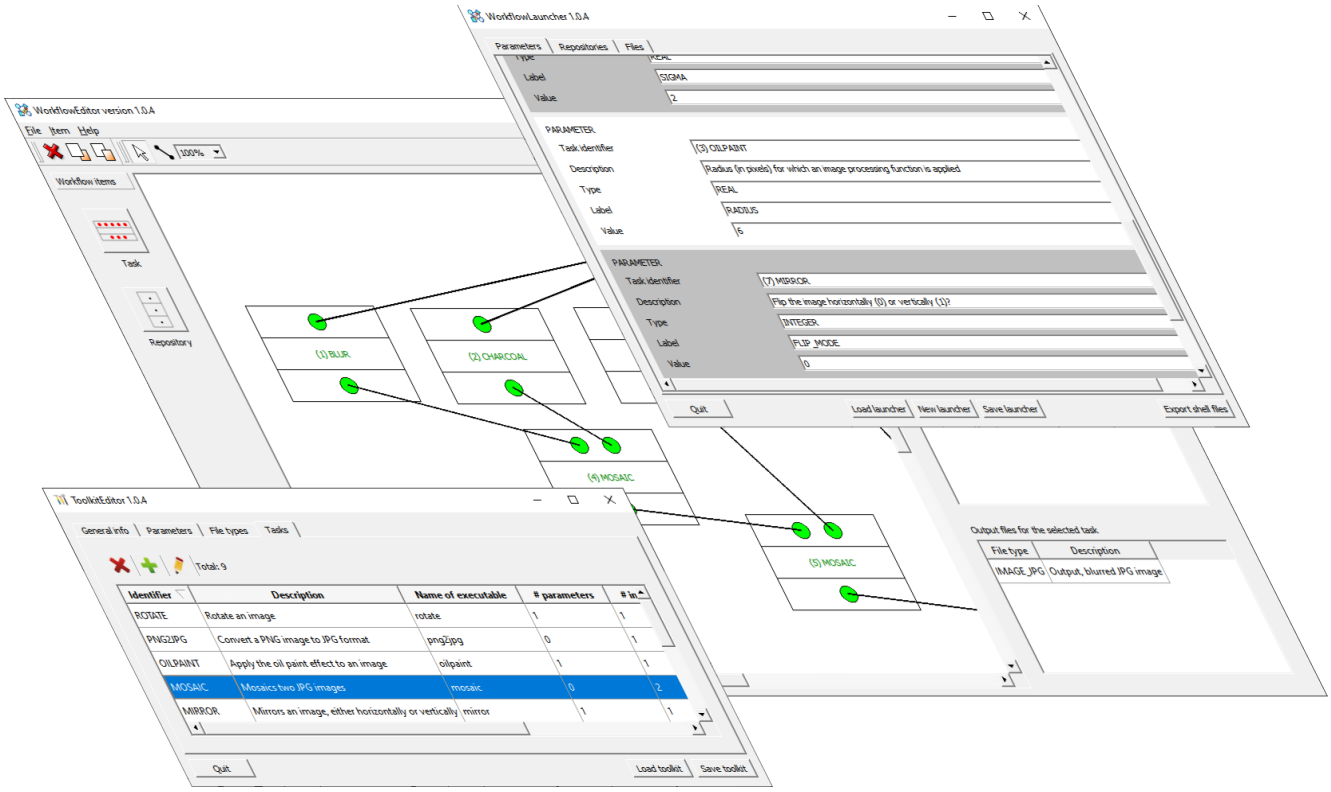
Building a WorkflowMaker-compliant console application

For WorkflowMaker version **1.1.0** and later
Windows 10 & 11, Ubuntu-based Linux distributions
64-bit only



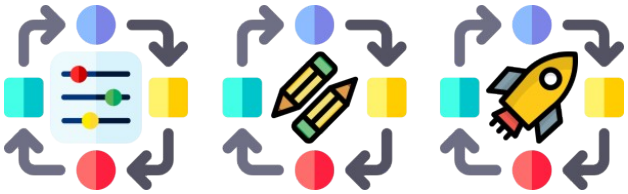


Table of contents

List of figures

1 The goal

This document explains, step by step, **how to build a WorkflowMaker-compliant console application** in C++. That is, the goal is to explain how to create an applications that may be integrated in WorkflowMaker toolkits / workflows / launchers. To do it, the charcoal application in the sample toolkit bundled with WorkflowMaker is used.

C++ is the language of choice for this tutorial; however, any other programming language able to create executable files might be used, providing that the restrictions set to the applications willing to be integrated in a WorkflowMaker toolkit are respected (see next section).

2 The restrictions to comply with

Although the restrictions that WorkflowMaker-compliant console applications are well documented in [RD1], these are repeated here for convenience reasons.

- They must be *console applications*, with no Graphical User Interface (GUI).
- All inputs, including the so-called keyboard parameters, must be files.
- All outputs must be files too.
- The keyboard parameters as well as the names of the input and output files must be defined in a single file, the *options file*.
- An options file must be written respecting a very simple format, consisting of lines made of pairs of labels and values (as, for instance, “RADIUS = 3.5”, without the quotes.).
- The application must be able to read this options file to retrieve the values of the keyboard parameters as well as the names of the input and output files.
- The application must be built as an executable file, that is, the operating system must be able to execute it without the intervention of any kind of middleware, such as Python interpreters or Java virtual machines. *This restriction may be eliminated in future versions of WorkflowMaker.*
- When the console application is run, it must accept a single command line parameter, defining the name of the options file mentioned above.
- They must return a status code stating whether it ended successfully (return code 0) or not (any other value).

For a more detailed explanation of these restrictions, refer to [RD1].

3 The steps to follow

According to [RD1], the steps to define a toolkit are these:

- Define the list of unique file types for all the toolkit.
- Define the list of unique keyboard parameters (again, for all the toolkit).
- Define the tasks, relying on the previous definitions.

In this section it will be assumed that the **list of unique file types** has already been defined after reviewing all the tasks that will become part of the image processing toolkit. These file types are summarized in Table 1:

Identifier	Description	Default extension
IMG_JPG	An image in .jpg format	.jpg
IMG_PNG	An image in .png format	.png

Table 1: The list of file types in the image processing example toolkit (source: [RD2])

That is, only two different kind of files will be handled by the toolkit: images in formats JPG and PNG.

Before proceeding, it is necessary to select one of the tools in the toolkit so the explanations may be as precise as necessary. The tool selected for this tutorial is **charcoal**. Any other tool in this toolkit (such as mosaic or rotate) might have been used, since all of them have been designed and implemented in a very similar way. However, charcoal is more convenient since it requires a set of keyboard parameters besides the usual input and output files, so the example will be complete.

The charcoal tool, as well as all the others in the image processing toolkit are documented in [RD2]. However, and for convenience reasons, some of its characteristics will be reproduced here.

Concerning the **list of keyboard parameters**, charcoal will contribute with two, namely the radius and standard deviation (in pixels) needed to apply a charcoal effect to the input image. The labels assigned to these parameters will be, respectively, **RADIUS** and **SIGMA**. Other tools will add more parameters to the list.

Note that assigning labels to parameters is a task that must be done **with the whole toolkit in mind.** Thus, if other tools in the toolkit (such as blur or oilpaint, see [RD2]) rely on conceptually identical parameters (radius and standard deviation of some kind) it is convenient to use the same labels in all tasks that rely on these. In this way the formal definition of these parameters (type, description) needs to be done only once per parameter using ToolkitEditor. Remember (see [RD1]) that a parameter defined in a toolkit, whatever its label, may be reused by as many tasks as necessary, providing actual values for these when a workflow is run).

Therefore, RADIUS and SIGMA are two labels that must be included in the options file that charcoal will read when executed.

To define the charcoal tasks we only need to define its input and output files. In the case of this tool, it takes an input JPG image and creates a new JPG one, to which the charcoal effect has been applied. Labels for these files must be selected. An obvious choice is **INPUT_FILENAME** and **OUTPUT_FILENAME** for, respectively, the input and output file names.

Said, that, to build a console application applying the charcoal effect parametrized by some radius and standard deviation to some input image and save the result to an output one, these are the tasks to

perform:

Write the code to read an options file with the following label / value pairs:

```
RADIUS          = <some value>
SIGMA           = <some other value>
INPUT_FILENAME  = <some file name>
OUTPUT_FILENAME = <some other file name>
```

where the text between brackets such as <some value> stand for actual value for the labels in the example. For instance, the following could be the contents of an options file for the charcoal tool:

```
RADIUS          = 5
SIGMA           = 8
INPUT_FILENAME  = image_to_process.jpg
OUTPUT_FILENAME = charcoaled_image.jpg
```

The developer is responsible for reading an options file including the set of labels defining the keyboard parameters and input / output files needed by the tool. However, these option files will be created automatically by WorkflowMaker, more precisely by the WorkflowLauncher tool, using the labels used when defining each task in the toolkit.

Write the logic for the tool.

Although said logic will vary between tasks, it must follow always the same pattern, that is:

- Retrieving the name of the options file name from the command line. This is the unique parameter accepted by the tool.
- Reading the options file, thus getting the values for its keyboard parameters and input / output files.
- Performing the actual process it has been devised for. In the case of the example tool, it is to apply the charcoal effect to the input image and save the result in an output one. The processing, however, will always imply reading one or more input files and saving one or more output ones.
- Return an error code stating the result of the process (see below).

Keeping track of errors. Report what happened.

To guarantee that workflows execute smoothly, the developer must keep track of the possible errors arising when the application is performing its task; for instance, in the case of the charcoal application, a negative value for the RADIUS parameter is unacceptable; if such condition is detected, an *nonzero* return code must be returned. The message is: make the tool robust handling the possible error

conditions arising and informing about these returning non-zero codes.

Note that in the case of a successful completion a zero must be returned.

Keeping a detailed tracking of errors is a very important task. When developing the task, **it is recommended to identify all the possible errors that might arise and assign a different error code to each of them.** WorkflowLauncher creates scripts to run workflows that print on screen the specific error code returned by a faulting application; if these error codes are specific enough, then the final user will be able to identify what caused the problem (assuming, of course, that the faulty application is properly documented and that it includes a list explaining the meaning of each error code!).

The following sections will show, by means of actual source code, how these steps have been implemented for the charcoal tool.

4 Interlude: the simple_options_file_parser library

Reading a label / value is so common a task when developing not only WorflowMaker-compliant console but also many other kinds of applications that using a library implementing this process is quite convenient.

This is the case of the image processing toolkit. A library, named simple_options_file_parser library has been included to facilitate this task. Please, refer to [RD3] for a detailed description about the options file format supported by this library. The documentation of its API in HTML format is available when building the image processing sample toolkit [RD4].

The source code of the charcoal tool relies on this library to read its options file. Developers are free to use said library at their convenience or use any other method to parse their options files.

5 The implementation of charcoal

The following sections will show in detail how the charcoal task has been implemented.

5.1 The files making the charcoal tool

The charcoal tool is composed of three header / source code files, namely:

charcoal.cpp – Implements the logic of the application, that is, it is responsible for getting the command line parameter, read the options file, apply the charcoal effect to the input image saving the result to a new output one. It also keeps track of (just some) error conditions, reporting what happened before finishing.

charcoal_options_file_reader.hpp / .cpp. Header and source code files, respectively, implementing an options file reader specific for the charcoal tool – that is, reading the set of label / value pairs it

needs. These files, as stated above, rely on the simple_options_file_parser library (see section 4).

There are other files in the folder. These are (1) mosaic.pro, the Qt project file to build the application and (2) postbuild.bat and postbuild.sh, scripts run after the tool has been compiled to perform maintenance task (such as creating folders or copying files to later building an installer or a deb package). However, these files are of no interest with regard to the explanations below.

5.2 Parsing the options

Although the developers willing to develop WorkflowMaker-compliant console applications are completely free to implement their own label / value option files parser, this section explains how this is done in the complete set of C++ source code samples included in the image processing example toolkit.

As stated in section 4, a very simple C++ library has been used to implement all tasks; this library provides a **generic** parser which just searches for the pattern “label = value” in the input file and stores a list of all the occurrences found. Afterwards, this parser must be interrogated to provide the values linked to the actual labels each task is interested in.

Obviously, the set of labels to ask for will vary from task to task, since each these will have defined its own labels corresponding to its keyboard parameters and input / output files.

Figure 1 (page 8) depicts the header (.hpp) file defining a specific parser for the charcoal task relying on said generic parser. In the figure, the **boldfaced red** items numbered from 1 to 3 correspond to the bullet points below:

1. As stated above, this parser class for the charcoal task relies on the simple_options_file_parser library. This #include statement loads all the necessary definitions to make available its resources.
2. This is the definition of a C++ struct named charcoal_options that will be used to store the options read from the options file.
3. This is the most important method in this class, the one that will parse the input options file whose name is set by parameter options_file and will set the values of a C++ struct of type charcoal_options (see previous bullet in this list) named options.

There are other methods defined in this figure, such as the constructor and destructor for objects of this class; others like build_error_list() and get_error_text() are provided to implement a mechanism able to return a message text explaining the meaning of the several error codes returned by method parse() (see bullet point 2 above), but these have no relevance with regard how the parsing process must be implemented by WorkflowMaker-compliant console applications.

```
#ifndef CHARCOAL_OPTIONS_FILE_READER_HPP
#define CHARCOAL_OPTIONS_FILE_READER_HPP

#include "simple_options_file_parser.hpp" // (1)

#include <string>
#include <vector>
#include <map>
#include <algorithm>
#include <iostream>

using namespace std;

struct charcoal_options                  // (2)
{
  string input_file_name;
  string output_filename;
  double radius;
  double sigma;
};

class charcoal_options_file_reader
{
  public:

            charcoal_options_file_reader  (void);

            ~charcoal_options_file_reader (void);

    string& get_error_text                (      int               error_code);

    int     parse_file                    (const string&          options_file,
                                           charcoal_options& options);       // (3)

  protected:

    void    build_error_list              (void);

  protected:

    map<int, string> error_messages_;
};

#endif // CHARCOAL_OPTIONS_FILE_READER_HPP
```

Figure 1: The header file defining the options file parser for the charcoal task.

If Figure 1 above shows the signatures of the methods defined by the charcoal's options file parser, Figure 2 (page 9) depicts the actual implementation of the parse() method, the one that actually retrieves the values of the several labels this task relies on. Again, there is a correspondence between the texts in **boldfaced red** and the bullet points in the next list.

```
int
charcoal_options_file_reader::
parse_file
(const string&     options_file,
 charcoal_options& options)                          // (1)
{
  {
    int                         error_line;
    simple_options_file_parser sofp;                 // (2)
    int                         status;
    string                      tstring;

    // Parse the requested options file.

    status = sofp.parse(options_file, error_line);  // (3)
    if (status != 0) return 1;

    //
    // Get all the options in the file, one by one. We'll try to
    // retrieve these using the proper types (int, double, etc.)
    // to check that they are correctly written in the file.
    //

    status = sofp.get_option_string ("INPUT_FILENAME",  options.input_file_name);  // (4)
    if (status != 0) return 2;

    status = sofp.get_option_string ("OUTPUT_FILENAME", options.output_filename);
    if (status != 0) return 3;

    status = sofp.get_option_double ("RADIUS",          options.radius);           // (5)
    if (status != 0) return 4;
    if (options.radius <= 0.0) return 4;

    status = sofp.get_option_double ("SIGMA",           options.sigma);
    if (status != 0) return 5;
    if (options.sigma <= 0.0) return 5;                                            // (6)

    // That's all!

    return 0;                                                                      // (7)
  }
}
```

Figure 2: Retrieving the necessary options from the options file.

In this figure:

1. These are the parameters received by the method in charge of parsing the options file. The first one, options_file, is the path and name of said file, while charcoal_options is a Ç++ struct (defined in charcoal_options_file_reader.pp) that will hold the values parsed by the method.

2. In this line a simple_options_file_parser object is declared and instantiated. This is the generic label / value pair parser used by all the tasks included in the image processing toolkit.

3. Here, said parser is used to actually parse the options file. The set of label / value pairs are stored internally by the generic label / value parser and will be retrieved afterwards using the API available for this library and class.

4. This line is retrieving the value corresponding to label INPUT_FILENAME. Since file names are stored as strings, the method used to retrieve this value is get_option_string(). There is still one more option (OUTPUT_FILENAME) that is retrieved using this method (see the lines below this one).

5. In this case, the option to retrieve (RADIUS) is a double value; therefore, the method used to load it is get_option_double(). Parameter SIGMA is retrieved in the same way (see the lines below this one).

6. In this line **a non-zero error code** is returned stating that something happened. This is crucial to let WorkflowMaker react when problems arise. Note that other non-zero error codes are returned in other places in this method (see the lines above this one).

7. Finally, and if no problems are detected, a zero return code is returned, indicating that everything worked as expected.

There are more methods to load values from the options file besides the ones shown in the figure above (get_option_string() and get_option_double()). See the API of the library simple_options_file_parser for more details.

Note that whenever an error condition may arise, an error code is returned. The ones highlighted in bullet points 6 and 7 above are just two examples. Reviewing the code will show that up to 6 different error conditions (1 to 5) and a success code are reported by this method. This will make much easier to identify what are the problems into which this application may run when a complaint is issued when running the script generated by WorkflowLauncher.

As stated at the beginning of this section, the header and implementation files shown in figures 1 and 2 are just a possible solution to parse label / value option files. These are provided as an example showing how to proceed using the simple_options_file_parser C++ library included along with the source code samples. Users are free to use any other implementation of their choice to implement this issue.

5.3 The main program

Figure 3 on page 11 shows the file charcoal.cpp, the file including the implementation for this task. Once more, the bullets in the numbered list below correspond to the **boldfaced red** text in said figure.

```
#include <string>
#include <iostream>
#include <Magick++.h>
#include "charcoal_options_file_reader.hpp"                              (1)

using namespace std;
using namespace Magick;

int
main
(int    argc,
 char** argv)
{
  {
    try
    {
      InitializeMagick(*argv);

      double                       charcoal_radius;
      double                       charcoal_sigma;
      string                       name_options_file;
      string                       name_the_input_image;
      string                       name_the_output_image;
      charcoal_options_file_reader op_reader;                            (2)
      charcoal_options             options;                              (3)
      int                          status;
      Image                        the_image;

      // Check that we've got the name of the options file.

      if (argc < 2) return 1; // No options file name in command line.   (4)

      // Read the options controlling our behaviour.

      name_options_file = argv[1];
      status = op_reader.parse_file(name_options_file, options);         (5)
      if (status != 0) return 2; // Error reading the options file.

      charcoal_radius       = options.radius;                            (6)
      charcoal_sigma        = options.sigma;
      name_the_input_image  = options.input_file_name;
      name_the_output_image = options.output_filename;

      // Read the images.

      the_image.read(name_the_input_image);

      // Apply the charcoal effect the image, according to the parameters read.

      the_image.charcoal (charcoal_radius, charcoal_sigma);

      // Write the image to disk.

      the_image.write(name_the_output_image);

      // That's all.

      return 0; // This error codes says "everything went right".        (7)
    }
    catch (...)
    {
       return 3; // Error when processing the image(s).                 (8)
    }
  }
}
```

Figure 3: The implementation of the charcoal task.

In this figure:

1. The class defined in figures 1 and 2 (the specific parser for charcoal's option files) is included and

2. a reader for these kind of files, op_reader, is instantiated; this reader will load the values of the labels

3. using a structure of type charcoal_options.
4. Since all WorkflowMaker-compliant console applications require a command line parameter, the name of the options file, we check that we have received it. If not, an error condition is reported.
5. If we have the name of the options file, then it is parsed. Again, if an error condition is reported by said parser, control is returned and the problem is properly indicated with another error code.
6. At this point, the options file has been loaded and its values assigned to local variables copied from the struct defined in (3).

Then, the actual process of applying the charcoal process takes place. Finally, two more exit conditions are tested:

7. The process finishes without errors; an error code (success) is returned.

8. Or, if the processing of the image throws an exception, another nonzero code is returned.

Other aspects regarding the code such as how the ImageMagick library is used in this example are not discussed here.

6 Conclusion

As seen in sections 5.1 to 5.3 the charcoal task (and all the others in the image processing example toolkit) implement the steps defined in section 3 (retrieve the name of the options file from the command line, read the options file, perform the specific task using the keyboard parameter and file name thus obtained and report the error or success conditions).

All of them are very simple examples of WorkflowMaker-compliant console applications; the rationale behind this simplicity was to avoid creating too much clutter, keeping the example as bare-bones as possible, so the essential principles on which said tasks rely were as evident as possible.

To finish, we would like to insist on the fact that the tasks in the image processing toolkit have been implemented C++, but that any programming language able to generate executable files providing a mechanism to retrieve parameters from the command line and returning error codes may be used instead.

Reference documents

All the documents listed below are part of **WorkflowMaker's documentation**.

[RD1] CTTC's Geomatics Research Unit (2024). WorkflowMaker user guide.
[RD2] CTTC's Geomatics Research Unit (2024). The image processing sample toolkit.
[RD3] CTTC's Geomatics Research Unit (2024). Simple options format file description.
[RD4] CTTC's Geomatics Research Unit (2024). Building the samples.