The Weaver Program

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

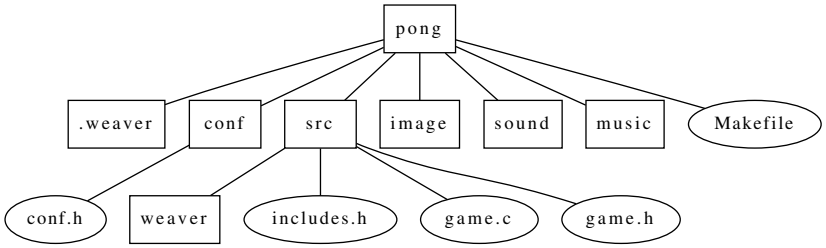
Abstract: *This article describes using literary programming the program Weaver. This program is a project manager for the Weaver Game Engine. If a user wants to create a new game with the Weaver Game Engine, they use this program to create the directory structure for a new game project. They also use this program to add new source files and shader files to a game project. And to update a project with a more recent Weaver version installed in the computer. The presenting code in C is cross-platform and should work under Windows, Linux, OpenBSD and possibly other Unix variants.*

1. Introduction

A game engine is made by a set of libraries and functions that helps a game creation offering common functionalities for this kind of development. But besides the libraries and functions, there should exist a manager responsible for creating some code which uses the library in a correct way and executes the necessary initializations.

The Weaver Game Engine has very strict prerequisites about how the directory with a game project should be organized. To follow these requisites, this program is necessary. It initializesin a correct way the directory structure in a new project. It adds new source files with the correct code to ensure the code integration. And controlling the project in this way, it also knows how to perform updates in the libraries for more recent versions.

This program usage is by the command line. For example, if a user types "`weaver pong`", a new directory structure like in the following image will be created.



The following sections in this document are organized in the following way. Section 2 is about this software license. Section 3 lists all the variables that control its execution. Section 4 defines some macros used in the program structure. Section 5 lists all the auxiliary functions defined. Sections 6 is how the variables are initialized. Section 7 is about the software use cases and how they are implemented after we have all the variables with the correct value.

2. Copyright and licensing

The software license is the GNU General Public License version 3:

Copyright (c) Thiago Leucz Astrizi 2015

```
This program is free software: you can redistribute it and/or
modify it under the terms of the GNU Affero General Public License as
published by the Free Software Foundation, either version 3 of
the License, or (at your option) any later version.

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 Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public
License along with this program.  If not, see
<http://www.gnu.org/licenses/>.
```

The complete version of the license can be obtained with the source code or checking the link above.

3. Variables and software structure

Weaver execution depends of the following variables:

`inside_weaver_directory` : If the program is invoked inside a Weaver project directory.

`argument` : The first argument, or NULL if doesn't exist.

`argument2` : The second argument, or NULL if doesn't exist.

`project_version_major` : If we are in a Weaver project, which is the major version number of the program which created the project? For example, if wea are in a project created by Weaver 0.5, the major version is 0. In tests version, the value is always 0.

`project_version_minor` : If we are in a Weaver project, the minor version number of the program which created the project. For example, if Weaver 0.5 created the current project, this number is 5. In test versions, the value is always 0.

`weaver_version_major` : The major version of this program.

`weaver_version_minor` : The minor version of this program.

`arg_is_path` : If the first argument exists and is an absolute or relative path in the filesystem.

`arg_is_valid_project` : If the first argument exists and would be considered a valid Weaver project name.

`arg_is_valid_module` : If the first argument exists and would be considered a valid module name in a Weaver project.

`arg_is_valid_plugin` : If the second argument exists and would be considered a valid plugin name in a Weaver project.

`arg_is_valid_function` : If the second argument exists and if it would be considered a valid name for a main loop and for a new file in a Weaver project.

`project_path` : If we are inside a Weaver project, which is the path for its base directory (where is the Makefile)?

`have_arg` : If the program is invoked with an argument.

`shared_dir` : The path to the directory where are the shared files from Weaver installation. The default is "`/usr/local/share/weaver`" in Unix systems and the "Program Files" folder in Windows. This can be changed in the program building defining the macro `WEAVER_DIR`.

`author_name` , `project_name` and `year` : The name of the user which is executing the program, the current project name (if we are inside a Weaver project directory) and the current year. This is important for copyright messages creation.

`return_value` : If the program is interrupted in this exact moment, what the program should return?

The software general structure with all the variables declarations is:

File: `src/weaver.c`:

```
<Section to be inserted: Headers Included in Weaver Program>
<Section to be inserted: Weaver Program Macros>
<Section to be inserted: Weaver Auxiliary Functions>
int main(int argc, char **argv){
  int return_value = 0; /* Return value. */
  bool inside_weaver_directory = false, arg_is_path = false,
    arg_is_valid_project = false, arg_is_valid_module = false,
    have_arg = false, arg_is_valid_plugin = false,
    arg_is_valid_function = false; /* Boolean variables. */
  unsigned int project_version_major = 0, project_version_minor = 0,
    weaver_version_major = 0, weaver_version_minor = 0,
    year = 0;
  /* Strings UTF-8: */
  char *argument = NULL, *project_path = NULL, *shared_dir = NULL,
    *author_name = NULL, *project_name = NULL, *argument2 = NULL;
<Section to be inserted: Initialization>
<Section to be inserted: Use Case 1: Printing help (create project)>
<Section to be inserted: Use Case 2: Printing management help>
<Section to be inserted: Use Case 3: Print version>
<Section to be inserted: Use Case 4: Updating Weaver project>
<Section to be inserted: Use Case 5: Create new module>
<Section to be inserted: Use Case 6: Create new project>
<Section to be inserted: Use Case 7: Create new plugin>
<Section to be inserted: Use Case 8: Create new shader>
<Section to be inserted: Use Case 9: Create new main loop>
END_OF_PROGRAM:
<Section to be inserted: Finishing>
  return return_value;
}
```

4. Macros and Headers in Weaver Program

This program needs some macros. The first one shall store a string with the program version; This version could be formed just by letters (if a test version) or by digits followed by a dot and more digits (without whitespaces) if this is a final version of the program.

For the second macro, observe that in the program structure above, exists a label called `END_OF_PROGRAM` in the finishing part. We can reach the label following the program normal execution, if nothing wrong happens. Otherwise, if an error happens, we can reach that label by an unconditional jump after printing the error message and adjusting the program return value. Treating this error condition with these actions is the second macro responsability.

We also could finish the program prematurely, but not because some error happened. The third macro will treat this case:

Section: Weaver Program Macros:

```
#define VERSION "Alpha"
#define W_ERROR() {perror(NULL); return_value = 1; goto END_OF_PROGRAM;}
#define END() goto END_OF_PROGRAM;
```

We are using the library function `perror`, so we need to include the header `stdio.h`, which will also bring us other useful functions to print in the screen or in files and to open and close files. We also should insert support for boolean values and the standard library with functions like `exit` utilized in the program structure:

Section: Headers Included in Weaver Program:

```
#include <stdio.h> // printf, fprintf, fopen, fclose, fgets, fgetc, perror
#include <stdbool.h> // bool, true, false
#include <stdlib.h> // free, exit, getenv
```

5. Auxiliary Functions

Here we list some functions which we should use in the program to facilitate its description.

5.1. path_up: Manipulating Paths

To manipulate directory tree paths, we define an auxiliary function which receives a path and erases the last characters until two "/" are erased. So in "/home/alice/project/dir/", it returns "/home/alice/project", goind one level up in the directory tree.

But in Windows systems, the separator isn't "/", but "". So we should treat the separator differently according with the Operating System:

Section: Weaver Auxiliary Functions:

```
void path_up(char *path){
#if !defined(_WIN32)
  char separator = '/';
#else
  char separator = '\\';
#endif
  int erased = 0;
  char *p = path;
  while(*p != '\0') p ++; // Vai at o fim
  while(erased < 2 && p != path){
    p --;
    if(*p == separator) erased ++;
    *p = '\0'; // Apaga
  }
}
```

Notice that if the function get a string without two separators, we erase all the string. In this program we will limit this function usage to strings with path for files outside the root directory, which are not the root directory themselves and for directories ended by the separator character. So we should always respect the limit of minimal two separators in paths. Example: "/etc/" and "/tmp/file.txt".

5.2. directory_exists: Arquivo existe e diretrio

To check if the directory `.weaver` exists, we define `directory_exist(x)` as a function which gets a file path and returns 1 if `x` is an existing directory, -1 if `x` is an existing file and 0 otherwise. Fist

let's create macros to make explicit the meaning of return values:

Section: Weaver Program Macros (continuation):

```
#define DONT_EXIST         0
#define EXISTS_AND_IS_DIR   1
#define EXISTS_AND_IS_FILE -1
```

Section: Weaver Auxiliary Functions (continuation):

```
int directory_exist(char *dir){
#if !defined(_WIN32)
  // Unix:
  struct stat s; // Stores if the file exists
  int err; // Checagem de erros
  err = stat(dir, &s); // It exists?
  if(err == -1) return DONT_EXIST;
  if(S_ISDIR(s.st_mode)) return EXISTS_AND_IS_DIR;
  return EXISTS_AND_IS_FILE;
#else
  // Windows:
  DWORD dwAttrib = GetFileAttributes(dir);
  if(dwAttrib == INVALID_FILE_ATTRIBUTES) return DONT_EXIST;
  if(!(dwAttrib & FILE_ATTRIBUTE_DIRECTORY)) return EXISTS_AND_IS_FILE;
  else return EXISTS_AND_IS_DIR;
#endif
}
```

Depending of the Operating System, we should utilize different functions and need different headers:

Section: Headers Included in Weaver Program:

```
#if !defined(_WIN32)
#include <sys/types.h> // stat, getuid, getpwuid, mkdir
#include <sys/stat.h> // stat, mkdir
#else
#include <windows.h> // GetFileAttributes, ...
#endif
```

5.3. concatenate: Concatenate strings

This function gets an arbitrary number of strings, but the last string must be `NULL` or the empty string. And it returns the concatenation of all the strings passed as argument. The function will always allocate a new string, which should be freed before the program ending.

Example: `concatenate("tes", " ", "t", "")` returns `"tes t"`.

Section: Weaver Auxiliary Functions (continuation):

```
char *concatenate(char *string, ...){
  va_list arguments;
  char *new_string, *current_string = string;
  size_t current_size = strlen(string) + 1;
  char *realloc_return;
  va_start(arguments, string);
  new_string = (char *) malloc(current_size);
  if(new_string == NULL) return NULL;
   // Copy the first string as recommended by the Operating System:
```

```
#ifdef __OpenBSD__
  strlcpy(new_string, string, current_size);
#else
  strcpy(new_string, string);
#endif
  while(current_string != NULL && current_string[0] != '\0'){
    current_string = va_arg(arguments, char *);
    current_size += strlen(current_string);
    realloc_return = (char *) realloc(new_string, current_size);
    if(realloc_return == NULL){
      free(new_string);
      return NULL;
    }
    new_string = realloc_return;
     // Copy the string as recommended by the Operating System:
#ifdef __OpenBSD__
    strlcat(new_string, current_string, current_size);
#else
    strcat(new_string, current_string);
#endif
  }
  return new_string;
}
```

This is a dangerous function that always should be invoked passing as last argument an empty string or NULL.

This function usage requires the following headers:

Section: Headers Included in Weaver Program:

```
#include <string.h> // strcmp, strcat, strcpy, strncmp
#include <stdarg.h> // va_start, va_arg
```

5.4. basename: Get a file name given its path

This function already exists in Unix systems. Given a complete path for a file, it returns a string with the file name. It doesn't neet to allocate a new string, it can just return a pointer for the filename inside the path string. Let's define it for Windows and other systems without a `basename` function:

Section: Weaver Auxiliary Functions (continuation):

```
#if defined(_WIN32)
char *basename(char *path){
  char *p = path;
  char *last_delimiter = NULL;
  while(*p != '\0'){
    if(*p == '\\')
      last_delimiter = p;
    p ++;
  }
  if(last_delimiter != NULL)
    return last_delimiter + 1;
  else
    return path;
}
```

```
#endif
```

In Unix Systems, we don't need to define this function, we just include its header:

Section: Headers Included in Weaver Program:

```
#if !defined(_WIN32)
#include <libgen.h>
#endif
```

5.5. copy_single_file: Copy single file to target directory

The function `copy_single_file` copies the file which path is the first argument to the target directory which path is the second argument. It returns 1 if successful or 0 otherwise.

Section: Weaver Auxiliary Functions (continuation):

```
int copy_single_file(char *file, char *directory){
  int block_size, bytes_read;
  char *buffer, *file_dst;
  FILE *orig, *dst;
  // Inicializa 'block_size':
                    <Section to be inserted: Discover block size>
  buffer = (char *) malloc(block_size); // Allocating buffer for copy
  if(buffer == NULL) return 0;
  file_dst = concatenate(directory, "/", basename(file), "");
  if(file_dst == NULL) return 0;
  orig = fopen(file, "r"); // Open origin file
  if(orig == NULL){
    free(buffer);
    free(file_dst);
    return 0;
  }
  dst = fopen(file_dst, "w"); // Open destiny file
  if(dst == NULL){
    fclose(orig);
    free(buffer);
    free(file_dst);
    return 0;
  }
  while((bytes_read = fread(buffer, 1, block_size, orig)) > 0){
    fwrite(buffer, 1, bytes_read, dst); // Copy origin to destiny
  }
  fclose(orig);
  fclose(dst);
  free(file_dst);
  free(buffer);
  return 1;
}
```

It's more efficient when the buffer used in the copy has the same size than a block in the filesystem. To get the correct valuem we use this code in Unix systems:

Section: Discover block size:

```
#if !defined(_WIN32)
{
  struct stat s;
```

```
  stat(directory, &s);
  block_size = s.st_blksize;
  if(block_size <= 0){
    block_size = 4096;
  }
}
#endif
```

In Windows we just assume that the size is 4KB:

Section: Discover block size (continuation):

```
#if defined(_WIN32)
  block_size = 4096;
#endif
```

5.6. copy_files: Copy all source files to destiny

With a function to copy a single file, we need to define a function to copy all the files inside a directory recursivelly. This requises some work, as we need to list all the content in a directory to get its files. How to do this depends of the Operating System.

In Unix systems we use the function `readdir` to read the content of directories:

Section: Weaver Auxiliary Functions (continuation):

```
#if !defined(_WIN32)
int copy_files(char *orig, char *dst){
  DIR *d = NULL;
  struct dirent *dir;
  d = opendir(orig);
  if(d){
    while((dir = readdir(d)) != NULL){ // Loop to read each file
      char *file;
      file = concatenate(orig, "/", dir -> d_name, "");
      if(file == NULL){
        return 0;
      }
#if (defined(__linux__) || defined(_BSD_SOURCE)) && defined(DT_DIR)
      // If we support DT_DIR, we don't need the funcion 'stat':
      if(dir -> d_type == DT_DIR){
#else
      struct stat s;
      int err;
      err = stat(file, &s);
      if(err == -1) return 0;
      if(S_ISDIR(s.st_mode)){
#endif
      // If we are dealing with a subdirectory:
        char *new_dst;
        new_dst = concatenate(dst, "/", dir -> d_name, "");
        if(new_dst == NULL){
          return 0;
        }
        if(strcmp(dir -> d_name, ".") && strcmp(dir -> d_name, "..")){
          if(directory_exist(new_dst) == DONT_EXIST) mkdir(new_dst, 0755);
```

```
          if(copy_files(file, new_dst) == 0){
            free(new_dst);
            free(file);
            closedir(d);
            return 0;
          }
        }
        free(new_dst);
      }
      else{
        // If we get a regular file:
        if(copy_single_file(file, dst) == 0){
          free(file);
          closedir(d);
          return 0;
        }
      }
    free(file);
    } // End of loop to read each file
    closedir(d);
  }
  return 1;
}
#endif
```

And this requires the following headers:

Section: Headers Included in Weaver Program:

```
#if !defined(_WIN32)
#include <dirent.h> // readdir, opendir, closedir
#endif
```

In Windows we don't need new headers. The function definition becames the following:

Section: Weaver Auxiliary Functions (continuation):

```
#if defined(_WIN32)
int copy_files(char *orig, char *dst){
  char *path, *search_path;
  WIN32_FIND_DATA file;
  HANDLE dir = NULL;
  search_path = concatenate(orig, "\\*", "");
  if(search_path == NULL)
    return 0;
  dir = FindFirstFile(search_path, &file);
  if(dir != INVALID_HANDLE_VALUE){
    // The first file shall be '.' and should be safely ignored
    do{
      if(strcmp(file.cFileName, ".") && strcmp(file.cFileName, "..")){
        path = concatenate(orig, "\\", file.cFileName, "");
        if(path == NULL){
          free(search_path);
          return 0;
        }
```

```
        if(file.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY){
          char *dst_path;
          dst_path = concatenate(dst, "\\", file.cFileName, "");
          if(directory_exist(dst_path) == DONT_EXIST)
            CreateDirectoryA(dst_path, NULL);
          if(copy_files(path, dst_path) == 0){
            free(dst_path);
            free(path);
            FindClose(dir);
            free(search_path);
            return 0;
          }
          free(dst_path);
        }
        else{ // file
          if(copy_single_file(path, dst) == 0){
            free(path);
            FindClose(dir);
            free(search_path);
            return 0;
          }
        }
        free(path);
      }
    }while(FindNextFile(dir, &file));
  }
  free(search_path);
  FindClose(dir);
  return 1;
}
#endif
```

5.7. write_copyright: Write copyright messages in files

By default Weaver projects are licensed under GNU GPL 3. As codes under this license are copied and utilized statically in new projects, the new projects needs the same license or a compatible one.

The code is very simple and requires just some parameters as the author name and the current year:

Section: Weaver Auxiliary Functions (continuation):

```
void write_copyright(FILE *fp, char *author_name, char *project_name, int year){
  char license[] = "/*\nCopyright (c) %s, %d\n\nThis file is part of %s.\n\n%s\
 is free software: you can redistribute it and/or modify\nit under the terms of\
 the GNU Affero General Public License as published by\nthe Free Software\
 Foundation, either version 3 of the License, or\n(at your option) any later\
 version.\n\n\
%s is distributed in the hope that it will be useful,\nbut WITHOUT ANY\
  WARRANTY; without even the implied warranty of\nMERCHANTABILITY or FITNESS\
  FOR A PARTICULAR PURPOSE.  See the\nGNU Affero General Public License for more\
  details.\n\nYou should have received a copy of the GNU Affero General Public License\
\nalong with %s. If not, see <http://www.gnu.org/licenses/>.\n*/\n\n";
```

```
  fprintf(fp, license, author_name, year, project_name, project_name,
          project_name, project_name);
}
```

5.8. create_dir: Create new directories

This function is responsible for creating a list of directories. This is a very simple thing, but should be encapsulated in a function because of differences between Operating Systems about how to do this task.

This function must receive as argument a variable number of strings, but the last argument must be an empty string or NULL. Earch argument except the last represents a path. The function will create the directory with the specified path. By default we use "/" as separator, so the function shall work both in Unix systems as in Windows. In the later, the function `CreateDirectoryA` accepts paths represented in Unix notation.

In Unix systems we need to specify the maximum permissions in the directory in terms os reading, writing and execution. The Operating System can then accept our recommended permissions, or ensure more restrictive ones depending of configuration. In Windows the permission logic is more hyerarquical, so we just use the same permissions as the parent directory.

In case of error, we return -1. Otherwise, we return 1.

The function definition is:

Section: Weaver Auxiliary Functions (continuation):

```
int create_dir(char *string, ...){
  char *current_string;
  va_list arguments;
  va_start(arguments, string);
  int err = 1;
  current_string = string;
  while(current_string != NULL && current_string[0] != '\0' && err != -1){
#if !defined(_WIN32)
    err = mkdir(current_string, S_IRWXU | S_IRWXG | S_IROTH);
#else
    if(!CreateDirectoryA(current_string, NULL))
      err = -1;
#endif
    current_string = va_arg(arguments, char *);
  }
  return err;
}
```

5.9. append_file: Concatenate file contents

This is an unusual function, it was designed to solve efficiently a single use case, sacrificing the consistency of its interface and its ease of use. It gets as argument a pointer for a target file already opened (usually we want to use this function after we opened a file to write the copyright notice), as second argument it gets the path of parent directory of an origin file and as the third argument it gets the origin file name.

Its definition is:

Section: Weaver Auxiliary Functions (continuation):

```
int append_file(FILE *fp, char *dir, char *file){
  int block_size, bytes_read;
  char *buffer, *directory = ".";
```

```
  char *path = concatenate(dir, file, "");
  if(path == NULL) return 0;
  FILE *origin;
```

<Section to be inserted: Discover block size>

```
  buffer = (char *) malloc(block_size);
  if(buffer == NULL){
    free(path);
    return 0;
  }
  origin = fopen(path, "r");
  if(origin == NULL){
    free(buffer);
    free(path);
    return 0;
  }
  while((bytes_read = fread(buffer, 1, block_size, origin)) > 0){
    fwrite(buffer, 1, bytes_read, fp);
  }
  fclose(origin);
  free(buffer);
  free(path);
  return 1;
}
```

6. Variable Initialization

6.1. inside_weaver_directory e project_path: Where we are

The first variable is `inside_weaver_directory`, which stores `false` if the program was invoked outside a Weaver project directory and `true` otherwise.

How should we detect if we are in a Weaver project directory? It's simple. They are directories which contains in them or in an ancestor directory a hidden directory named `.weaver`. If we find this directory, we can also adjust the variable `project_path` to point to where is this directory. If we don't find it, we are outside a Weaver directory and we don't need to change these variables default value, which are `false` and `NULL`.

In short, we need a loop with the following characteristics:

Invariant: The variable `complete_path` must always store the complete path of the directory `.weaver` if this file hypothetically existed in the current directory.

Initialization: We initialize `complete_path` to be valid when we are in or initial current directory.

Maintenance: In each iteration we check if we found a termination condition. If not, we change to the current directory parent, always updating the variables to keep valid the invariant.

Termination: We terminate the loop if one of the following 3 conditions occur:

a) `complete_path == "/.weaver"` : We can't go to a parent directory because we are already in the root of the filesystem. It means that we aren't in a Weaver directory.

b) `complete_path == "C:\\.weaver"` : In fact, the initial letter could be "D", "E" or any other letter, not just "C". It also could be ""weaver". This means that we are in the root of a Windows filesystem (the last case without a drive letter represents a network directory) and we aren't in a Weaver directory.

c) `complete_path == "./.weaver"` and this file exists and is a directory: In this case, we were inside a Weaver directory. We can also update `project_path` to store the current path.

The initialization of these variables is then:

Section: Initialization:

```
char *path = NULL, *complete_path = NULL;
#if !defined(_WIN32)
```

```
path = getcwd(NULL, 0); // Unix
#else
{ // Windows
  DWORD bsize;
  bsize = GetCurrentDirectory(0, NULL);
  path = (char *) malloc(bsize);
  GetCurrentDirectory(bsize, path);
}
#endif
if(path == NULL) W_ERROR();
complete_path = concatenate(path, "/.weaver", "");
free(path);
if(complete_path == NULL) W_ERROR();
```

To get the current directory, we need the header:

Section: Headers Included in Weaver Program:

```
#if !defined(_WIN32)
#include <unistd.h> // get_current_dir_name, getcwd, stat, chdir, getuid
#endif
```

Now we define the described loop:

Section: Initialization (continuation):

```
{
  // Testa se chegamos ao fim:
  while(strcmp(complete_path, "/.weaver") &&
        strcmp(complete_path, "\\.weaver") &&
        strcmp(complete_path + 1, ":\\.weaver")){
    if(directory_exist(complete_path) == EXISTS_AND_IS_DIR){
      inside_weaver_directory = true;
      complete_path[strlen(complete_path) - 7] = '\0'; // Apaga o '.weaver'
      project_path = concatenate(complete_path, "");
      if(project_path == NULL){ free(complete_path); W_ERROR(); }
      break;
    }
    else{
      path_up(complete_path);
#ifdef __OpenBSD__
      {
        size_t tmp_size = strlen(complete_path);
        strlcat(complete_path, "/.weaver", tmp_size);
      }
#else
      strcat(complete_path, "/.weaver");
#endif
    }
  }
  free(complete_path);
}
```

We allocated memory to `project_path`, so in the end of program we need to free this memory:

Section: Finishing:

```
if(project_path != NULL) free(project_path);
```

6.2. weaver_version_major e weaver_version_minor: Program Version

To discover the current program version, we can just check the macro `VERSION`. Then we get the majorand minor version number parsing the digits separated by a dot (if they exist). If we can't find a dot in the version name, this is a test version and the minor and major version number must be treated as 0. So we can just use `atoi` function and this requirement will be fullfilled:

Section: Initialization (continuation):

```
{
  char *p = VERSION;
  while(*p != '.' && *p != '\0') p ++;
  if(*p == '.') p ++;
  weaver_version_major = atoi(VERSION);
  weaver_version_minor = atoi(p);
}
```

6.3. project_version_major e project_version_minor: Project Version

If we are inside a Weaver project, we need to initialize these variables with Weaver major and minor version used to create the project, or update the project if it was updated in the past. This can be obtained checking the file *.weaver/version* inside the Weaver directory. If we aren't in a Weaver directory, we don't need to get these values. The major and minor version usually is separated by a dot, following the same rules than in previous subsection.

Section: Initialization (continuation):

```
if(inside_weaver_directory){
  FILE *fp;
  char *p, version[10];
  char *file_path = concatenate(project_path, ".weaver/version", "");
  if(file_path == NULL) W_ERROR();
  fp = fopen(file_path, "r");
  free(file_path);
  if(fp == NULL) W_ERROR();
  p = fgets(version, 10, fp);
  if(p == NULL){ fclose(fp); W_ERROR(); }
  while(*p != '.' && *p != '\0') p ++;
  if(*p == '.') p ++;
  project_version_major = atoi(version);
  project_version_minor = atoi(p);
  fclose(fp);
}
```

6.4. have_arg, argument e argument2: Invocation Arguments

The easiest to initialize variables. We just check for `argc` and `argv`.

Section: Initialization (continuation):

```
have_arg = (argc > 1);
if(have_arg) argument = argv[1];
if(argc > 2) argument2 = argv[2];
```

6.5. arg_is_path: If the argument is a directory

If we have a first argument, we need to check if it's a path for a directory where is a Weaver project. For this, we just concatenate `/.weaver` in the first argument and check if this file exist.

Section: Initialization (continuation):

```
if(have_arg){
  char *buffer = concatenate(argument, "/.weaver", "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) == EXISTS_AND_IS_DIR){
    arg_is_path = 1;
  }
  free(buffer);
}
```

6.6. shared_dir: Where the files are installed

The variable `shared_dir` shall contain where are the installed shared files. These files are libraries to be inserted statically and models of source code. If the macro `WEAVER_DIR` exists because it was defined during compilation, this will be the path where are these files. Other wise, se use `/usr/local/share/weaver` in Unix systems and `%PROGRAMFILES%\weaver` in Windows systems.

Section: Initialization (continuation):

```
{
#ifdef WEAVER_DIR
  shared_dir = concatenate(WEAVER_DIR, "");
#else
#if !defined(_WIN32)
  shared_dir = concatenate("/usr/local/share/weaver/", ""); // Unix
#else
  { // Windows
    char *temp_buf = NULL;
    DWORD bsize = GetEnvironmentVariable("ProgramFiles", temp_buf, 0);
    temp_buf = (char *) malloc(bsize);
    GetEnvironmentVariable("ProgramFiles", temp_buf, bsize);
    shared_dir = concatenate(temp_buf, "\\weaver\\", "");
    free(temp_buf);
  }
#endif
#endif
  if(shared_dir == NULL) W_ERROR();
}
```

With this code we allow the user to choose the install directory during compilation. This usually is more common in Unix systems than in Windows where programs are expected to be kept in the same place.

In Windows the code is longer because we need to determine manually where to store the files. The path can change depending of system language, drive unit, or if the program is 32 or 64 bits.

In both cases, during program finalization, we need to free the memory allocated to `shared_dir` :

Section: Finishing (continuation):

```
if(shared_dir != NULL) free(shared_dir);
```

6.7. arg_is_valid_project: If the argument is a project name

The next question is if what we got as argument, if we got something, could be a valid Weaver project name or not. To answer this, three conditions need to be fullfilled:

1) The project basename must be formed only by alphanumeric characters and underline (a "/" or ""." can appear in the path, not in the basename).

2) A file with the same name can't already exist in the place indicated for the new project.

3) The project can't have the same name as some file that exists in the base directory of a Weaver project ("Makefile", for example). Otherwise, during compilation we would overwrite important files.

For this, we use the following code:

Section: Initialization (continuation):

```
if(have_arg && !arg_is_path){
  char *buffer;
  char *base = basename(argument);
  int size = strlen(base);
  int i;
  // Checking for invalid characters
  for(i = 0; i < size; i ++){
    if(!isalnum(base[i]) && base[i] != '_'){
      goto NOT_VALID;
    }
  }
  // Checking if file exists:
  if(directory_exist(argument) != DONT_EXIST){
    goto NOT_VALID;
  }
  // Checking for name conflicts:
  buffer = concatenate(shared_dir, "project/", base, "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) != DONT_EXIST){
    free(buffer);
    goto NOT_VALID;
  }
  free(buffer);
  arg_is_valid_project = true;
}
NOT_VALID:
```

To check for alphanumeric characters, we include the following header:

Section: Headers Included in Weaver Program:

```
#include <ctype.h> // isalnum
```

6.8. arg_is_valid_module: If the argument could be a module name

Checking if the first argument could be a valid module name makes sense only if we are inside a Weaver directory (otherwise we wouldn't care about modules) and if we have a first argument. In this case the argument is a valid module name if it's formed by just alphanumeric characters, underline and if doesn't exist a file with the same name and extension `.c`

or .h in src/:

Section: Initialization (continuation):

```
if(have_arg && inside_weaver_directory){
  char *buffer;
  int i, size;
  size = strlen(argument);
  // Checando caracteres invlidos no nome:
  for(i = 0; i < size; i ++){
    if(!isalnum(argument[i]) && argument[i] != '_'){
      goto NOT_VALID_MODULE;
    }
  }
  // Checking name conflicts:
  buffer = concatenate(project_path, "src/", argument, ".c", "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) != DONT_EXIST){
    free(buffer);
    goto NOT_VALID_MODULE;
  }
  buffer[strlen(buffer) - 1] = 'h';
  if(directory_exist(buffer) != DONT_EXIST){
    free(buffer);
    goto NOT_VALID_MODULE;
  }
  free(buffer);
  arg_is_valid_module = true;
}
NOT_VALID_MODULE:
```

6.9. arg_is_valid_plugin: If the argument could be a plugin name

An argument is a valid plugin name if it's formed only by alphanumeric characters and underline and if doesn't exist in the directory `plugin` a file with the same name and extension `.c` or `.h`. We also must be inside a Weaver directory.

Section: Initialization (continuation):

```
if(argument2 != NULL && inside_weaver_directory){
  int i, size;
  char *buffer;
  size = strlen(argument2);
  // Checando caracteres invlidos no nome:
  for(i = 0; i < size; i ++){
    if(!isalnum(argument2[i]) && argument2[i] != '_'){
      goto NOT_VALID_PLUGIN;
    }
  }
  // Checando se j existe plugin com mesmo nome:
  buffer = concatenate(project_path, "plugins/", argument2, ".c", "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) != DONT_EXIST){
    free(buffer);
```

```
    goto NOT_VALID_PLUGIN;
  }
  free(buffer);
  arg_is_valid_plugin = true;
}
NOT_VALID_PLUGIN:
```

6.10. arg_is_valid_function: If the second argument could be the name of a main loop

This variable will be true if exists a second argument formed only by alphanumeric characters or underline. And also the first character must me a letter and it can't have the same name than a reserved word in C or C++. The last checked versions for these languages is the draft C++20 and C11.

Section: Initialization (continuation):

```
if(argument2 != NULL && inside_weaver_directory &&
   !strcmp(argument, "--loop")){
  int i, size;
  char *buffer;
  // First character isn't digit
  if(isdigit(argument2[0]))
    goto NOT_VALID_FUNCTION;
  size = strlen(argument2);
  // Checking invalid characters
  for(i = 0; i < size; i ++){
    if(!isalnum(argument2[i]) && argument2[i] != '_'){
      goto NOT_VALID_PLUGIN;
    }
  }
  // Checking if file already exist
  buffer = concatenate(project_path, "src/", argument2, ".c", "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) != DONT_EXIST){
    free(buffer);
    goto NOT_VALID_FUNCTION;
  }
  buffer[strlen(buffer)-1] = 'h';
  if(directory_exist(buffer) != DONT_EXIST){
    free(buffer);
    goto NOT_VALID_FUNCTION;
  }
  free(buffer);
  // Checking reserved words
  const char *reserved[] = {"alignas", "alignof", "and", "and_eq",
                            "asm", "auto", "bitand", "bitor", "bool",
                            "break", "case", "catch", "char", "char8_t",
                            "char16_t", "char32_t", "class", "compl",
                            "concept", "const", "consteval", "constexpr",
                            "constinit", "const_cast", "continue",
                            "co_await", "co_return", "co_yield",
                            "decltype", "default", "delete", "do",
```

```
                              "double", "dynamic_cast", "else", "enum",
                              "explicit", "export", "extern", "false",
                              "float", "for", "friend", "goto", "if",
                              "inline", "int", "long", "mutable",
                              "namespace", "new", "noexcept", "not",
                              "not_eq", "nullptr", "operator", "or",
                              "or_eq", "private", "protected", "public",
                              "register", "reinterpret_cast", "requires",
                              "restrict", "return", "short", "signed",
                              "sizeof", "static", "static_assert",
                              "static_cast", "struct", "switch", "template",
                              "this", "thread_local", "throw", "true",
                              "try", "typedef", "typeid", "typename",
                              "union", "unsigned", "using", "virtual",
                              "void", "volatile", "xor", "xor_eq",
                              "wchar_t", "while", NULL};
  for(i = 0; reserved[i] != NULL; i ++)
    if(!strcmp(argument2, reserved[i]))
      goto NOT_VALID_FUNCTION;
  arg_is_valid_function = true;
}
NOT_VALID_FUNCTION:
```

6.11. author_name: Code creator name

The variable `author_name` must contain the name of the user which is invoking the program. This information is useful to generate copyright notices in source files of new modules.

Initialize this information is different in Unix and Windows systems. In Unix systems we begin getting their UID. Then we get login information with `getpwuid`. If the user has a registered name in `/etc/passwd`, we use this name. Otherwise, we just use the login name.

Section: Initialization (continuation):

```
#if !defined(_WIN32)
{
  struct passwd *login;
  int size;
  char *string_to_copy;
  login = getpwuid(getuid()); // Get user data
  if(login == NULL) W_ERROR();
  size = strlen(login -> pw_gecos);
  if(size > 0)
    string_to_copy = login -> pw_gecos;
  else
    string_to_copy = login -> pw_name;
  size = strlen(string_to_copy);
  author_name = (char *) malloc(size + 1);
  if(author_name == NULL) W_ERROR();
#ifdef __OpenBSD__
  strlcpy(author_name, string_to_copy, size + 1);
#else
  strcpy(author_name, string_to_copy);
#endif
```

```
}
#endif
```

In Windows the name can be obtained with the function `GetUserNameExA` . In the first invocation we try to get the buffer size to store the name and in the second we get the name. In case of error, we try the more ancient function `GetUserNameA` which will return a simpler username instead of the full name. And in this case, we allocate in the buffer space to store the biggest valid username.

Section: Initialization (continuation):

```
#if defined(_WIN32)
{
  int size = 0;
  GetUserNameExA(NameDisplay, author_name, &size);
  if(GetLastError() == ERROR_MORE_DATA){
    if(size == 0)
      size = 64;
    author_name = (char *) malloc(size);
    if(GetUserNameExA(NameDisplay, author_name, &size) == 0){
      size = UNLEN + 1;
      author_name = (char *) malloc(size);
      GetUserNameA(author_name, &size);
    }
  }
  else{
    size = UNLEN + 1;
    author_name = (char *) malloc(size);
    GetUserNameA(author_name, &size);
  }
}
#endif
```

After we need to free the memory allocated to `author_name` :

Section: Finishing (continuation):

```
if(author_name != NULL) free(author_name);
```

For this code work, we need to insert the correct library according with the Operating System. In Unix is `pwd.h` to get `getpwuid` and in Windows we insert the security header.

Section: Headers Included in Weaver Program:

```
#if !defined(_WIN32)
#include <pwd.h> // getpwuid
#else
#define SECURITY_WIN32
#include <Security.h>
#include <Lmcons.h>
#endif
```

6.12. project_name: The project name

Talking about the project name makes sense only when we are in a Weaver project directory. In this case, the project name can be found in the file `.weaver/name` inside the base directory:

Section: Initialization (continuation):

```
if(inside_weaver_directory){
```

```
  FILE *fp;
  char *c;
#if !defined(_WIN32)
  char *filename = concatenate(project_path, ".weaver/name", "");
#else
  char *filename = concatenate(project_path, ".weaver\name", "");
#endif
  if(filename == NULL) W_ERROR();
  project_name = (char *) malloc(256);
  if(project_name == NULL){
    free(filename);
    W_ERROR();
  }
  fp = fopen(filename, "r");
  if(fp == NULL){
    free(filename);
    W_ERROR();
  }
  c = fgets(project_name, 256, fp);
  fclose(fp);
  free(filename);
  if(c == NULL) W_ERROR();
  project_name[strlen(project_name)-1] = '\0';
  project_name = realloc(project_name, strlen(project_name) + 1);
  if(project_name == NULL) W_ERROR();
}
```

After we need to free the memory allocated to `project_name` :

Section: Finishing (continuation):

```
if(project_name != NULL) free(project_name);
```

6.13. year: Current year

The current year can be obtained with the function `localtime` in any Operating System:

Section: Initialization (continuation):

```
{
  time_t current_time;
  struct tm *date;
  time(&current_time);
  date = localtime(&current_time);
  year = date -> tm_year + 1900;
}
```

The prerequisite is include the headers with time functions:

Section: Headers Included in Weaver Program:

```
#include <time.h> // localtime, time
```

7. Use Cases

7.1. Printing help about project creation

The first use case happens when Weaver is called outside a Weaver project directory and

the invocation is without arguments or with the argument `--help`. In this case, we assume that the user doesn't know entirelly how to use the program and print a help message. The message will be something like this:

```
    .  .   You are outside a Weaver Directory.
   ./  \.  The following command uses are available:
    \\  //
    \\()//  weaver
    .={}=.      Print this message and exits.
  / /‘’\ \
  ‘ \  / ’  weaver PROJECT_NAME
     ‘’         Creates a new Weaver Directory with a new
                project.
```

This is made with the following code:

Section: Use Case 1: Printing help (create project):

```
if(!inside_weaver_directory && (!have_arg || !strcmp(argument, "--help"))){
  printf("    .  .     You are outside a Weaver Directory.\n"
  "    .|  |.    The following command uses are available:\n"
  "    ||  ||\n"
  "    \\\\()//  weaver\n"
  "    .={}=.      Print this message and exits.\n"
  "  / /‘’\\ \\\n"
  "  ‘ \\  / ’  weaver PROJECT_NAME\n"
  "     ‘’        Creates a new Weaver Directory with a new\n"
  "               project.\n");
  END();
}
```

7.2. Printing help about project management

The second use case is also very simple. It happens when we are already in a Weaver directory and we call Weaver without arguments or with a `--help`. We assume in this case that the user wants instructions about creation of new modules or other parts of a project. The printed message will be:

```
       \              You are inside a Weaver Directory.
        \______/      The following command uses are available:
        /\____/\
       / /\__/\ \       weaver
    __/_/_/\/\_\_\___     Prints this message and exits.
      \ \ \/\/ / /
       \ \/__\/ /       weaver NAME
        \/____\/          Creates NAME.c and NAME.h, updating
        /      \          the Makefile and headers
       /
                         weaver --loop NAME
                          Creates a new main loop in a new file src/NAME.c

                         weaver --plugin NAME
                          Creates new plugin in plugin/NAME.c

                         weaver --shader NAME
                          Creates a new shader directory in shaders/
```

We get this with the code:

Section: Use Case 2: Printing management help:

```
if(inside_weaver_directory && (!have_arg || !strcmp(argument, "--help"))){
  printf("       \\                You are inside a Weaver Directory.\n"
  "         \\______/        The following command uses are available:\n"
  "         /\\____/\\\n"
  "        / /\\__/\\ \\       weaver\n"
  "    __/_/_/\\/\\_\\_\\___     Prints this message and exits.\n"
  "      \\ \\ \\/\\/ / /\n"
  "       \\ \\/__\\/ /       weaver NAME\n"
  "        \\/____\\/          Creates NAME.c and NAME.h, updating\n"
  "        /      \\          the Makefile and headers\n"
  "       /\n"
  "                         weaver --loop NAME\n"
  "                          Creates a new main loop in a new file src/NAME.c\n\n"
  "                         weaver --plugin NAME\n"
  "                          Creates a new plugin in plugin/NAME.c\n\n"
  "                         weaver --shader NAME\n"
  "                          Creates a new shader directory in shaders/\n");
  END();
}
```

7.3. Showing the Weaver version

A use case even simpler. It happens when a user calls Weaver with the argument `--version`:

Section: Use Case 3: Print version:

```
if(have_arg && !strcmp(argument, "--version")){
  printf("Weaver\t%s\n", VERSION);
  END();
}
```

7.4. Updating existing Weaver projects

This use case happens when the user pass as argument an absolute or relative path to a Weaver directory. We assume in this case that the user wants toupdate the project passed as argument.Perhaps it was created with an older version and they want to update to a newer version.

Naturally, this will be done only if the Weaver version installed in the machine is equal or higher than the version used in the project. Or if we have installed a test version. In the later case, we assume that the user wants to try the experimental version. Ignoring test versions, it's not possible to make downgrades, going from 0.2 version to 0.1.

If the project version and the installed version is the same, we still update. This is a useful way to corrrect a project if someone erases or corrupt an important source file in `src/weaver/`.

Experimental versions always are identified as having a name formed only by alphabetic characters ("Alpha" ou "Beta", for example). Stable versions are formed by one or more digits and a dot followed by one or more digits (the majr and minor version). As the version numbers are interpreted using `atoi` , if we are using an experimental version, we identify the major and minor version as 0.

Projects in experimental versions always are updated, even if the installed version is older than the project experimental version.

Updating consists in copying all the files in the Weaver shared directory in `project/src/weaver` to

the directory `src/weaver` inside the project. For this we use the auxiliary functions defined in previous sections.

Section: Use Case 4: Updating Weaver project:

```
if(arg_is_path){
  if((weaver_version_major == 0 && weaver_version_minor == 0) ||
     (weaver_version_major > project_version_major) ||
     (weaver_version_major == project_version_major &&
      weaver_version_minor >= project_version_minor)){
    char *buffer, *buffer2;
    // |buffer| <- SHARED_DIR/project/src/weaver
    buffer = concatenate(shared_dir, "project/src/weaver/", "");
    if(buffer == NULL) W_ERROR();
    // |buffer2| <- PROJECT_DIR/src/weaver/
    buffer2 = concatenate(argument, "/src/weaver/", "");
    if(buffer2 == NULL){
      free(buffer);
      W_ERROR();
    }
    if(copy_files(buffer, buffer2) == 0){
      free(buffer);
      free(buffer2);
      W_ERROR();
    }
    free(buffer);
    free(buffer2);
  }
  END();
}
```

7.5. Adding a new module in a Weaver project

If we are inside a Weaver project directory and we got an argument, we assume that this argument is a new module for our game. If the argument is a valid module name, we create the new module with this name. Otherwise we print an error message and exit.

Creating a new module involves:

a) Creating base files `.c` and `.h`, giving them the same name as the new module.

b) Adding in both files a copyright and licensing comment with author name, project name and current year.

c) Adding in `.h` macro code to prevent the header of being inserted more than one time and including the module header in the file with `.c` extension.

d) Including the module header with extension `.h` in the file `src/includes.h` to ensure that its functions and structures can be acessed from all the other source files.

The code for this is:

Section: Use Case 5: Create new module:

```
if(inside_weaver_directory && have_arg &&
   strcmp(argument, "--plugin") && strcmp(argument, "--shader") &&
   strcmp(argument, "--loop")){
  if(arg_is_valid_module){
    char *filename;
    FILE *fp;
    // Creating module.c
    filename = concatenate(project_path, "src/", argument, ".c", "");
```

```
    if(filename == NULL) W_ERROR();
    fp = fopen(filename, "w");
    if(fp == NULL){
      free(filename);
      W_ERROR();
    }
    write_copyright(fp, author_name, project_name, year);
    fprintf(fp, "#include \"%s.h\"", argument);
    fclose(fp);
    filename[strlen(filename)-1] = 'h'; // Creating module.h
    fp = fopen(filename, "w");
    if(fp == NULL){
      free(filename);
      W_ERROR();
    }
    write_copyright(fp, author_name, project_name, year);
    fprintf(fp, "#ifndef _%s_h_\n", argument);
    fprintf(fp, "#define _%s_h_\n\n#include \"weaver/weaver.h\"\n",
            argument);
    fprintf(fp, "#include \"includes.h\"\n\n#endif");
    fclose(fp);
    free(filename);
    // Updating src/includes.h to insert modulo.h:
    fp = fopen("src/includes.h", "a");
    fprintf(fp, "#include \"%s.h\"\n", argument);
    fclose(fp);
  }
  else{
    fprintf(stderr, "ERROR: This module name is invalid.\n");
    return_value = 1;
  }
  END();
}
```

7.6. Creating a new project

Creating a new Weaver project involves creating a directory with the project name, copying to there all the content of directory `project` in Weaver shared directory and creating a directory `.weaver` with the project data. We also create a `src/game.c` and `src/game.h` adding the copyright comment there and copying inside them the basic structure from the files `basefile.c` and `basefile.h`. We also create an empy `src/includes.h` to be written in the future when new modules will be created.

Section: Use Case 6: Create new project:

```
if(! inside_weaver_directory && have_arg){
  if(arg_is_valid_project){
    int err;
    char *dir_name;
    FILE *fp;
    err = create_dir(argument, NULL);
    if(err == -1) W_ERROR();
#if !defined(_WIN32)
    err = chdir(argument);
```

```
#else
    err = _chdir(argument);
#endif
    if(err == -1) W_ERROR();
    err = create_dir(".weaver", "conf", "tex", "src", "src/weaver",
                     "fonts", "image", "sound", "models", "music",
                     "plugins", "src/misc", "src/misc/sqlite",
                     "compiled_plugins", "shaders", "");
    if(err == -1) W_ERROR();
    dir_name = concatenate(shared_dir, "project", "");
    if(dir_name == NULL) W_ERROR();
    if(copy_files(dir_name, ".") == 0){
      free(dir_name);
      W_ERROR();
    }
    free(dir_name); // Creating file with version number
    fp = fopen(".weaver/version", "w");
    fprintf(fp, "%s\n", VERSION);
    fclose(fp); // Creating file with project name
    fp = fopen(".weaver/name", "w");
    fprintf(fp, "%s\n", basename(argv[1]));
    fclose(fp);
    fp = fopen("src/game.c", "w");
    if(fp == NULL) W_ERROR();
    write_copyright(fp, author_name, argument, year);
    if(append_file(fp, shared_dir, "basefile.c") == 0) W_ERROR();
    fclose(fp);
    fp = fopen("src/game.h", "w");
    if(fp == NULL) W_ERROR();
    write_copyright(fp, author_name, argument, year);
    if(append_file(fp, shared_dir, "basefile.h") == 0) W_ERROR();
    fclose(fp);
    fp = fopen("src/includes.h", "w");
    write_copyright(fp, author_name, argument, year);
    fprintf(fp, "\n#include \"weaver/weaver.h\"\n");
    fprintf(fp, "\n#include \"game.h\"\n");
    fclose(fp);
  }
  else{
    fprintf(stderr, "ERROR: %s is not a valid project name.", argument);
    return_value = 1;
  }
  END();
}
```

7.7. Creating a new plugin

This use case is invoked when we have two arguments, the first one is `"--plugin"` and the second one is the new plugin name, which can't conflict with the name of an already existing plugin in the directory `plugins/`. We must be inside a Weaver project directory to create a plugin:

Section: Use Case 7: Create new plugin:

```
if(inside_weaver_directory && have_arg && !strcmp(argument, "--plugin") &&
   arg_is_valid_plugin){
  char *buffer;
  FILE *fp;
  /* Creating the file: */
  buffer = concatenate("plugins/", argument2, ".c", "");
  if(buffer == NULL) W_ERROR();
  fp = fopen(buffer, "w");
  if(fp == NULL) W_ERROR();
  write_copyright(fp, author_name, project_name, year);
  fprintf(fp, "#include \"../src/weaver/weaver.h\"\n\n");
  fprintf(fp, "void _init_plugin_%s(W_PLUGIN){\n\n}\n\n", argument2);
  fprintf(fp, "void _fini_plugin_%s(W_PLUGIN){\n\n}\n\n", argument2);
  fprintf(fp, "void _run_plugin_%s(W_PLUGIN){\n\n}\n\n", argument2);
  fprintf(fp, "void _enable_plugin_%s(W_PLUGIN){\n\n}\n\n", argument2);
  fprintf(fp, "void _disable_plugin_%s(W_PLUGIN){\n\n}\n", argument2);
  fclose(fp);
  free(buffer);
  END();
}
```

7.8. Creating a new shader

This use case is similar to the previous one, but with some differences. All shader will be a new file in GLSL format inside the directory `shaders`. Also their names will always be in the format of regular expression `[0-9][0-9]*-.*`. The digit(s) in the firszt part must be unique for each shader in the same project. And the numbers must always be sequential, starting in 1 and incrementing with each new shader.

This use case will be invoked when our first argument is `''--shader''` and the second one is any name. We don't need to require names in a particular format because the numeric restrictions over the initial digits ensure that each shader will have a unique and nonconflicting name.

To ensure this restriction, the code will count the number of files with GLSL extension in the shader directory and create a new shader with the name `DD-XX.glsl`, where `DD` is the number of existing files plus 1 and `XX` is the name chosen as the second argument got by the program. But if we find holes in the shader numbering, for example if exists a shader 3 without a shader 2, we will choode `DD` to fill the hole.

After setting the shader numbering, we create it as an empty file and copy the content from an existing file in the shared directory where Weaver is installed.

Depois de descobrir a numerao do novo shader, basta criarmos ele como um arquivo vazio e depois copiarmos o contedo de um modelo j existente em nosso diretrio de instalao.

The code of this use case is:

Section: Use Case 8: Create new shader:

```
if(inside_weaver_directory && have_arg && !strcmp(argument, "--shader") &&
   argument2 != NULL){
    FILE *fp;
    size_t tmp_size, number = 0;
    char *buffer;
     <Section to be inserted: Shader: Count number of files and get shader number>
    // Creating the shader:
    tmp_size = number / 10 + 7 + strlen(argument2);
    buffer = (char *) malloc(tmp_size);
    if(buffer == NULL) W_ERROR();
    buffer[0] = '\0';
```

```
    snprintf(buffer, tmp_size, "%d-%s.glsl", (int) number, argument2);
    fp = fopen(buffer, "w");
    if(fp == NULL){
        free(buffer);
        W_ERROR();
    }
    if(append_file(fp, shared_dir, "shader.glsl") == 0) W_ERROR();
    fclose(fp);
    free(buffer);
    END();
}
```

The part where we count the number of files is different in Unix and in Windows because of different API to deal with the filesystem. What we will do is iterate over the files in the directory `shaders` and for each file which isn't a directory, has the extension GLSL and has a name which starts with a positive number, we mark its number as true in a boolean vector initially set as entirely false. After iterating over all files we check the boolean vector searching for the first position marked as false. This position is a number still not used for a shader and its index is our chosen number.

Aditional complexities involves esuring that our boolean vector has a good size. We initially allocate it with 128 positions, but if we find a shader with bigger number, we realloc it to deal with the biggger number.

The code for this in Linux is:

Section: Shader: Count number of files and get shader number:

```
#if !defined(_WIN32)
{
  size_t i, max_number = 0;
  DIR *shader_dir;
  struct dirent *dp;
  char *p;
  bool *exists;
  size_t exists_size = 128;
  shader_dir = opendir("shaders/");
  if(shader_dir == NULL)
    W_ERROR();
  exists = (bool *) malloc(sizeof(bool) * exists_size);
  if(exists == NULL){
    closedir(shader_dir);
    W_ERROR();
  }
  for(i = 0; i < exists_size; i ++)
    exists[i] = false;
  while((dp = readdir(shader_dir)) != NULL){
    if(dp -> d_name == NULL) continue;
    if(dp -> d_name[0] == '.') continue;
    if(dp -> d_name[0] == '\0') continue;
    buffer = concatenate("shaders/", dp -> d_name, "");
    if(buffer == NULL) W_ERROR();
    if(directory_exist(buffer) != EXISTS_AND_IS_FILE){
      free(buffer);
      continue;
    }
```

```
    for(p = buffer; *p != '\0'; p ++);
    p -= 5;
    if(strcmp(p, ".glsl") && strcmp(p, ".GLSL")){
      free(buffer);
      continue;
    }
    number = atoi(buffer);
    if(number == 0){
      free(buffer);
      continue;
    }
    if(number > max_number)
      max_number = number;
    if(number > exists_size){
      if(number > exists_size * 2)
        exists_size = number;
      else
        exists_size *= 2;
      exists = (bool *) realloc(exists, exists_size * sizeof(bool));
      if(exists == NULL){
        free(buffer);
        closedir(shader_dir);
        W_ERROR();
      }
      for(i = exists_size / 2; i < exists_size; i ++)
        exists[i] = false;
    }
    exists[number - 1] = true;
    free(buffer);
  }
  closedir(shader_dir);
  for(i = 0; i <= max_number; i ++)
    if(exists[i] == false){
      break;
    }
  free(exists);
}
#endif
```

In Windows the code to deal with files is different, but the other parts are the same:

Section: Shader: Count number of files and get shader number:

```
#if defined(_WIN32)
{
  int i;
  char *p;
  bool *exists;
  size_t exists_size = 128;
  int number, max_number = 0;
  WIN32_FIND_DATA file;
  HANDLE shader_dir = NULL;
  shader_dir = FindFirstFile("shaders\\", &file);
```

```
if(shader_dir == INVALID_HANDLE_VALUE)
  W_ERROR();
exists = (bool *) malloc(sizeof(bool) * exists_size);
if(exists == NULL){
  FindClose(shader_dir);
  W_ERROR();
}
for(i = 0; i < exists_size; i ++)
  exists[i] = false;
do{
  if(file.cFileName == NULL) continue;
  if(file.cFileName[0] == '.') continue;
  if(file.cFileName[0] == '\0') continue;
  buffer = concatenate("shaders\\", file.cFileName, "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) != EXISTS_AND_IS_FILE){
    free(buffer);
    continue;
  }
  for(p = buffer; *p != '\0'; p ++);
  p -= 5;
  if(strcmp(p, ".glsl") && strcmp(p, ".GLSL")){
    free(buffer);
    continue;
  }
  number = atoi(buffer);
  if(number == 0){
    free(buffer);
    continue;
  }
  if(number > max_number)
    max_number = number;
  if(number > exists_size){
    if(number > exists_size * 2)
      exists_size = number;
    else
      exists_size *= 2;
    exists = (bool *) realloc(exists, exists_size * sizeof(bool));
    if(exists == NULL){
      free(buffer);
      FindClose(shader_dir);
      W_ERROR();
    }
    for(i = exists_size / 2; i < exists_size; i ++)
      exists[i] = false;
  }
  exists[number - 1] = true;
  free(buffer);
}while(FindNextFile(shader_dir, &file) != 0);
FindClose(shader_dir);
for(i = 0; i <= max_number; i ++)
```

```
  if(exists[i] == false){
    break;
  }
  free(exists);
}
#endif
```

7.9. Creating a new main loop

This use case happens when the second argument is `--loop` and when the next argument is a valid name for a C and C++ function. If it's not, we print an error message to warn the user.

In this case we can't just copy the content of a base file to our new file as we did previously in the code for a new module, because we need to define a function with the given name in the file. So we create the file and write inside it the content hard-coded in the following code:

Section: Use Case 9: Create new main loop:

```
if(inside_weaver_directory && !strcmp(argument, "--loop")){
  if(!arg_is_valid_function){
    if(argument2 == NULL)
      fprintf(stderr,
              "ERROR: You should pass a name for your new loop.\n");
    else
      fprintf(stderr, "ERROR: %s not a valid loop name.\n", argument2);
    W_ERROR();
  }
  char *filename;
  FILE *fp;
  // Creating LOOP_NAME.c
  filename = concatenate(project_path, "src/", argument2, ".c", "");
  if(filename == NULL) W_ERROR();
  fp = fopen(filename, "w");
  if(fp == NULL){
    free(filename);
    W_ERROR();
  }
  write_copyright(fp, author_name, project_name, year);
  fprintf(fp, "#include \"%s.h\"\n\n", argument2);
  fprintf(fp, "MAIN_LOOP %s(void){\n", argument2);
  fprintf(fp, " LOOP_INIT:\n\n");
  fprintf(fp, " LOOP_BODY:\n");
  fprintf(fp, "  if(W.keyboard[W_ANY])\n");
  fprintf(fp, "    Wexit_loop();\n");
  fprintf(fp, " LOOP_END:\n");
  fprintf(fp, "  return;\n");
  fprintf(fp, "}\n");
  fclose(fp);
  // Creating LOOP_NAME.h
  filename[strlen(filename)-1] = 'h';
  fp = fopen(filename, "w");
  if(fp == NULL){
    free(filename);
```

```
    W_ERROR();
  }
  write_copyright(fp, author_name, project_name, year);
  fprintf(fp, "#ifndef _%s_h_\n", argument2);
  fprintf(fp, "#define _%s_h_\n#include \"weaver/weaver.h\"\n\n", argument2);
  fprintf(fp, "#include \"includes.h\"\n\n");
  fprintf(fp, "MAIN_LOOP %s(void);\n\n", argument2);
  fprintf(fp, "#endif\n");
  fclose(fp);
  free(filename);
  // Updating src/includes.h
  fp = fopen("src/includes.h", "a");
  fprintf(fp, "#include \"%s.h\"\n", argument2);
  fclose(fp);
}
```

8. Conclusion

This end the necessary code to create the Weaver program, a manager of eaver projects.

The progam described here, in fact, doesn't contain everything to manage a project. There's somo work too in the Weaver installer, which in Unix is a Makefile and in Windows systems is a MSIX package.

The Weaver library code also is part of the Weaver engine, but will have their code described in other articles.

There's also some base code for shaders and new projects which should be find in the directory `project` in the source code.

And finally there's also the complex code of Makefiles and Window's `.vcxproj` files, where more management code can be found.