O Programa Weaver

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.*

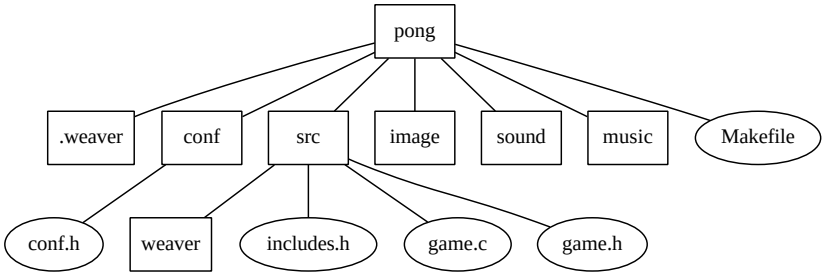
Resumo: *Este artigo descreve usando programação literária o programa Weaver. Este programa é um gerenciador de projetos para o Motor de Jogos Weaver. Se alguém deseja criar um novo projeto com o motor de jogos, usará este programa para criar a estrutura de diretórios desejada. Também usará o programa para adicionar novos arquivos de código-fonte e shaders. Para atualizar um projeto pré-existente com uma nova versão de Weaver, o programa também é necessário. O código seguinte em C será multi-plataforma e deverá funcionar em Windows, Linux, OpenBSD e possivelmente outras variantes de Unix.*

1. Introdução

Um motor de jogos é formado por um conjunto de bibliotecas e funções que auxiliam na criação de jogos fornecendo as funcionalidades mais comus para este tipo de desenvolvimento. Mas além das bibliotecas e funções, deve existir um gerenciador responsável por fazer com que o seu código utilize as bibliotecas a maneira adequada e faça as inicializações necessárias.

O motor de jogos Weaver tem pré-requisitos bastante estritos de como o diretório que contém um projeto Weaver deve estar organizado. É para cumprir erstes requisitos que o programa que será apresentado é necessário. Ele inicializa da maneira correta a estrutura de diretórios de um novo projeto. Ele adiciona novos arquivos fonte já com quaisquer código necessário para sua integração. E por controlar o projeto desta forma, ele saberá atualizar as bibliotecas para versões mais recentes se necessário.

O uso deste programa será por mieo de linha de comando. Por exemplo, se um usuário usar o comando "`weaver pong`", será criada uma estrutura de diretórios semelhante à mostrada na imagem que ilustra o fim da seção com um novo projeto chamado "pong".



As seguintes seções do artigo estão organizadas da seguinte forma. A seção 2 abordará a licensa do software. A seção 3 listará as variáveis usadas para controlar seu comportamento. A seção 4 trará algumas macros que usaremos, algumas das quais apareceram na estrutura do programa. A seção 5 apresentará algumas funções auxiliares que utilizaremos. A seção 6 mostrará a inicialização das variáveis do programa. A seção 7 mostrará os casos de uso do programa e como implementá-los após termos as variáveis com os valores certos.

2. Copyright e licenciamento

Segue abaixo a licença do programa e sua tradução não-oficial:

```
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/>.
```

```
Copyright (c) Thiago Leucz Astrizi 2015

Este programa é um software livre; você pode redistribuí-lo e/ou
modificá-lo dentro dos termos da Licença Pública Geral GNU Affero como
publicada pela Fundação do Software Livre (FSF); na versão 3 da
Licença, ou (na sua opinião) qualquer versão.

Este programa é distribuído na esperança de que possa ser útil,
mas SEM NENHUMA GARANTIA; sem uma garantia implícita de ADEQUAÇÃO
a qualquer MERCADO ou APLICAÇÃO EM PARTICULAR. Veja a
Licença Pública Geral GNU Affero para maiores detalhes.

Você deve ter recebido uma cópia da Licença Pública Geral GNU Affero
junto com este programa. Se não, veja
```

<http://www.gnu.org/licenses/>.

A versão completa da licença pode ser obtida junto ao código-fonte Weaver ou consultada no link mencionado.

3. Variáveis e Estrutura do Programa Weaver

O comportamento de Weaver deve depender das seguintes variáveis:

`inside_weaver_directory` : Indicará se o programa está sendo invocado de dentro de um projeto Weaver.

`argument` : O primeiro argumento, ou NULL se ele não existir

`argument2` : O segundo argumento, ou NULL se não existir.

`project_version_major` : Se estamos em um projeto Weaver, qual o maior número da versão do Weaver usada para gerar o projeto. Exemplo: se a versão for 0.5, o número maior é 0. Em versões de teste, o valor é sempre 0.

`project_version_minor` : Se estamos em um projeto Weaver, o valor do menor número da versão do Weaver usada para gerar o projeto. Exemplo, se a versão for 0.5, o número menor é 5. Em versões de teste o valor é sempre 0.

`weaver_version_major` : O número maior da versão do Weaver sendo usada no momento.

`weaver_version_minor` : O número menor da versão do Weaver sendo usada no momento.

`arg_is_path` : Se o primeiro argumento é ou não um caminho absoluto ou relativo para um projeto Weaver.

`arg_is_valid_project` : Se o argumento passado seria válido como nome de projeto Weaver.

`arg_is_valid_module` : Se o argumento passado seria válido como um novo módulo no projeto Weaver atual.

`arg_is_valid_plugin` : Se o segundo argumento existe e se ele é um nome válido para um novo plugin.

`arg_is_valid_function` : Se o segundo argumento existe e se ele seria um nome válido para um loop principal e também para um arquivo.

`project_path` : Se estamos dentro de um diretório de projeto Weaver, qual o caminho para a sua base (onde há o Makefile)

`have_arg` : Se o programa é invocado com argumento.

`shared_dir` : Deverá armazenar o caminho para o diretório onde estão os arquivos compartilhados da instalação de Weaver. Por padrão, será igual à "`/usr/local/share/weaver`" em sistemas Unix e dentro de subpasta em "Arquivos de Programas" no Windows. Isso pode ser mudado durante a compilação definindo a macro `WEAVER_DIR`.

`author_name` , `project_name` e `year` : Conterão respectivamente o nome do usuário que está invocando Weaver, o nome do projeto atual (se estivermos no diretório de um) e o ano atual. Isso será importante para gerar as mensagens de Copyright em novos projetos Weaver.

`return_value` : Que valor o programa deve retornar caso o programa seja interrompido no momento atual.

A estrutura geral do programa com a declaração de todas as variáveis será:

Arquivo: `src/weaver.c`**:**

<Seção a ser Inserida: Cabeçalhos Incluídos no Programa Weaver>
<Seção a ser Inserida: Macros do Programa Weaver>
<Seção a ser Inserida: Funções auxiliares Weaver>

```
int main(int argc, char **argv){
  int return_value = 0; /* Valor de retorno. */
  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; /* Variáveis booleanas. */
  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;
                    <Seção a ser Inserida: Inicialização>
     <Seção a ser Inserida: Caso de uso 1: Imprimir ajuda (criar projeto)>
    <Seção a ser Inserida: Caso de uso 2: Imprimir ajuda de gerenciamento>
            <Seção a ser Inserida: Caso de uso 3: Mostrar versão>
         <Seção a ser Inserida: Caso de uso 4: Atualizar projeto Weaver>
           <Seção a ser Inserida: Caso de uso 5: Criar novo módulo>
           <Seção a ser Inserida: Caso de uso 6: Criar novo projeto>
           <Seção a ser Inserida: Caso de uso 7: Criar novo plugin>
           <Seção a ser Inserida: Caso de uso 8: Criar novo shader>
         <Seção a ser Inserida: Caso de uso 9: Criar novo loop principal>
END_OF_PROGRAM:
                      <Seção a ser Inserida: Finalização>
  return return_value;
}
```

4. Macros e Cabeçalhos do Programa Weaver

O programa precisará de algumas macros. A primeira delas deverá conter uma string com a versão do programa. A versão pode ser formada só por letras (no caso de versões de teste) ou por um número seguido de um ponto e de outro número (sem espaços) no caso de uma versão final do programa.

Para a segunda macro, observe que na estrutura geral do programa vista acima existe um rótulo chamado `END_OF_PROGRAM` logo na parte de finalização. Uma das formas de chegarmos lá é por meio da execução normal do programa, caso nada dê errado. Entretanto, no caso de um erro, nós podemos também chegar lá por meio de um desvio incondicional após imprimirmos a mensagem de erro e ajustarmos o valor de retorno do programa. A responsabilidade de fazer isso será da segunda macro.

Por outro lado, podemos também querer encerrar o programa previamente, mas sem que tenha havido um erro. A responsabilidade disso é da terceira macro que definimos.

Seção: Macros do Programa Weaver:

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

Como estamos usando a função de biblioteca `perror`, devemos incluir o cabeçalho `stdio.h`, o que também nos trará s funções de imprimir na tela, abrir e fechar arquivos e escrever neles, o que nos será útil. Vamos inserir suporte à valores booleanos que usamos na própria estrutura do programa e também a biblioteca padrão, que tem funções como `exit` usadas na estrutura do programa:

Seção: Cabeçalhos Incluídos no Programa Weaver:

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

```
#include <stdlib.h> // free, exit, getenv
```

5. Funções Auxiliares

Listemos aqui algumas funções que usaremos ao longo do programa para facilitar sua descrição.

5.1. path_up: Manipula Caminho

Para manipularmos o caminho da árvore de diretórios, usaremos uma função auxiliar que recebe como entrada uma string com um caminho na árvore de diretórios e apaga todos os últimos caracteres até apagar dois "/". Assim em "/home/alice/projeto/diretorio/" ele retornaria "/home/alice/projeto" efetivamente subindo um nível na árvore de diretórios.

É importante lembrar que no Windows o separador não é o "/", mas o """. Então vamos tratar o separador de forma diferente de acordo com o sistema:

Seção: Funções auxiliares Weaver:

```
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 até o fim
  while(erased < 2 && p != path){
    p --;
    if(*p == separator) erased ++;
    *p = '\0'; // Apaga
  }
}
```

Note que caso a função receba uma string que não possua dois "/" em seu nome, acabamos apagando toda a string. Neste programa limitaremos o uso desta função a strings com caminhos de arquivos que não estão na raíz e diretórios diferentes da própria raíz que terminam sempre com "/", então não teremos problemas pois a restrição do número de barras será cumprida. Ex: "/etc/" e "/tmp/file.txt".

5.2. directory_exists: Arquivo existe e é diretório

Para checar se o diretório `.weaver` existe, definimos `directory_exist(x)` como uma função que recebe uma string correspondente à localização de um arquivo e que deve retornar 1 se `x` for um diretório existente, -1 se `x` for um arquivo existente e 0 caso contrário. Primeiro criamos as macros para não nos esquecermos do que significa cada número de retorno:

Seção: Macros do Programa Weaver (continuação):

```
#define NAO_EXISTE             0
#define EXISTE_E_EH_DIRETORIO  1
#define EXISTE_E_EH_ARQUIVO   -1
```

Seção: Funções auxiliares Weaver (continuação):

```
int directory_exist(char *dir){
#if !defined(_WIN32)
  // Unix:
  struct stat s; // Armazena status se um diretório existe ou não.
  int err; // Checagem de erros
```

```
  err = stat(dir, &s); // .weaver existe?
  if(err == -1) return NAO_EXISTE;
  if(S_ISDIR(s.st_mode)) return EXISTE_E_EH_DIRETORIO;
  return EXISTE_E_EH_ARQUIVO;
#else
  // Windows:
  DWORD dwAttrib = GetFileAttributes(dir);
  if(dwAttrib == INVALID_FILE_ATTRIBUTES) return NAO_EXISTE;
  if(!(dwAttrib & FILE_ATTRIBUTE_DIRECTORY)) return EXISTE_E_EH_ARQUIVO;
  else return EXISTE_E_EH_DIRETORIO;
#endif
}
```

Dependendo de estarmos no Windows ou em sistemas Unix, usamos funções diferentes e vamos precisar de cabeçalhos diferentes:

Seção: Cabeçalhos Incluídos no Programa Weaver:

```
#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: Concatena strings

Esta função deve receber um número arbitrário de srings como argumento, mas a última string deve ser uma string vazia ou `NULL`. E irá retornar a concatenação de todas as strings passadas como argumento. A função irá alocar sempre uma nova string, a qual deverá ser desalocada antes do programa terminar. Como exemplo, `concatenate("tes", " ", "te", "")` retorna `"tes te"`.

Seção: Funções auxiliares Weaver (continuação):

```
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;
   // Copia primeira string de acordo com o indicado pelo sistema operacional
#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;
```

```
    // Copia próxima string de acordo com o recomendado pelo sistema
#ifdef __OpenBSD__
    strlcat(new_string, current_string, current_size);
#else
    strcat(new_string, current_string);
#endif
  }
  return new_string;
}
```

É importante lembrarmos que a função `concatenate` sempre deve receber como último argumento uma string vazia ou teremos um *buffer overflow*. Esta função é perigosa e deve ser usada sempre tomando-se este cuidado.

O uso desta função requer que usemos o seguinte cabeçalho:

Seção: Cabeçalhos Incluídos no Programa Weaver:

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

5.4. basename: Obtém o nome de um arquivo dado seu caminho

Esta função já existe em sistemas Unix. Dado o caminho completo para um arquivo, ela retorna uma string apenas com o nome do arquivo. Ela não precisa alocar uma nova string, ela retorna um ponteiro para o nome do arquivo dentro do próprio caminho recebido como argumento. Vamos defini-la agora para sistemas Windows:

Seção: Funções auxiliares Weaver (continuação):

```
#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
```

Mesmo que não precisemos definir esta função em sistemas Unix, ainda precisamos incluí-la com o cabeçalho:

Seção: Cabeçalhos Incluídos no Programa Weaver:

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

5.5. copy_single_file: Copia um único arquivo para diretório

A função `copy_single_file` tenta copiar o arquivo cujo caminho é o primeiro argumento para o diretório cujo caminho é o segundo argumento. Se ela conseguir, retorna 1 e retorna 0 caso contrário.

Seção: Funções auxiliares Weaver (continuação):

```
int copy_single_file(char *file, char *directory){
  int block_size, bytes_read;
  char *buffer, *file_dst;
  FILE *orig, *dst;
  // Inicializa 'block_size':
  <Seção a ser Inserida: Descobre tamanho do bloco do sistema de arquivos>
  buffer = (char *) malloc(block_size); // Aloca buffer de cópia
  if(buffer == NULL) return 0;
  file_dst = concatenate(directory, "/", basename(file), "");
  if(file_dst == NULL) return 0;
  orig = fopen(file, "r"); // Abre arquivo de origem
  if(orig == NULL){
    free(buffer);
    free(file_dst);
    return 0;
  }
  dst = fopen(file_dst, "w"); // Abre arquivo de destino
  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); // Copia origem -> buffer -> destino
  }
  fclose(orig);
  fclose(dst);
  free(file_dst);
  free(buffer);
  return 1;
}
```

O mais eficiente é que o buffer usado para copiar arquivos tenha o mesmo tamanho do bloco do sistema de arquivos. Para obter o valor correto deste tamanho, usamos o seguinte trecho de código em sistemas Unix:

Seção: Descobre tamanho do bloco do sistema de arquivos:

```
#if !defined(_WIN32)
{
  struct stat s;
  stat(directory, &s);
  block_size = s.st_blksize;
  if(block_size <= 0){
    block_size = 4096;
  }
}
#endif
```

No Windows nós apenas iremos assumir que o tamanho é 4 KB:

Seção: Descobre tamanho do bloco do sistema de arquivos (continuação):

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

5.6. copy_files: Copia todos os arquivos de origem para destino

De posse da função que copia um só arquivo, precisamos definir uma função para copiar todos os arquivos de dentro d eum diretório para outro recursivamente. Isso é algo trabalhoso, pois precisamos listar todo o conteúdo de um diretório para obter seus arquivos. Como fazer isso varia dependendo do sistema operacional.

Em sistemas Unix a função usará `readdir` para ler o conteúdo de arquivos:

Seção: Funções auxiliares Weaver (continuação):

```
#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 para ler cada arquivo
      char *file;
      file = concatenate(orig, "/", dir -> d_name, "");
      if(file == NULL){
        return 0;
      }
#if (defined(__linux__) || defined(_BSD_SOURCE)) && defined(DT_DIR)
      // Se suportamos DT_DIR, não precisamos chamar a função '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
      // Se concluirmos estar lidando com subdiretório via 'stat' ou 'DT_DIR':
        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) == NAO_EXISTE) mkdir(new_dst, 0755);
          if(copy_files(file, new_dst) == 0){
            free(new_dst);
            free(file);
            closedir(d);
            return 0;
          }
        }
        free(new_dst);
      }
      else{
        // Se concluimos estar diante de um arquivo usual:
        if(copy_single_file(file, dst) == 0){
          free(file);
          closedir(d);
          return 0;
        }
```

```
      }
    free(file);
    } // Fim do loop para ler cada arquivo
    closedir(d);
  }
  return 1;
}
#endif
```

E isso requer inserir o cabeçalho:

Seção: Cabeçalhos Incluídos no Programa Weaver:

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

No Windows, não é necessário inserir nenhum cabeçalho novo que já não está inserido. A definição da função fica assim:

Seção: Funções auxiliares Weaver (continuação):

```
#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) == NAO_EXISTE)
            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: Escreve mensagem de copyright em arquivo

Por padrão, projetos Weaver utilizam a licença GNU GPL3. Como códigos sob esta licença são copiados e usados estaticamente em novos projetos, eles precisam necessariamente ter uma licença igual ou compatível.

O código é bastante simples e requer apenas alguns parâmetros com o nome do autor e ano atual:

Seção: Funções auxiliares Weaver (continuação):

```
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: Cria novos diretórios

Esta é a função responsável por criar uma lista de diretórios. Isso é algo bastante simples, mas deve ficar encapsulado em sua própria função por causa das diferenças entre Sistemas Operacionais sobre como realizar a tarefa.

Esta função deve receber uma lista variável de strings como argumento, sendo que o último argumento precisa ser uma string vazia ou NULL. Para cada argumento representando um caminho, a função criará o diretório no caminho especificado. Por padrão, usaremos como separador em caminhos será o "/", de modo que isso fará a função funcionar tanto em sistemas Unix como no Windows, onde a sua função `CreateDirectoryA` entende caminhos passados com o separador Unix.

Em sistemas Unix nós precisamos também especificar as permissões máximas que o diretório terá em termos de leitura, escrita e execução. Tais permissões podem ser mais restritas de acordo com a configuração do sistema. No Windows, que tem um sistema de permissões mais hierárquica, nós apenas herdamos as permissões do diretório pai.

Em caso de erro, nós retornaremos -1. Se tudo deu certo, retornamos 1.

A definição de nossa função será então:

Seção: Funções auxiliares Weaver (continuação):

```
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: Concatena um arquivo no outro

Esta será uma função diferente, pois ela será mais focada em resolver de maneira eficiente um único caso de uso do que apresentar uma interface intuitiva e consistente. Ela irá receber como entrada um ponteiro para um arquivo já aberto (iremos chamar esta função depois de já termos aberto o arquivo para escrever o copyright) e receberá dois outros argumentos: o diretório em que está o arquivo de origem e o nome do arquivo de origem, separados em duas strings. Dessa forma, o trabalho de concatenar as duas coisas fica com esta função, não com quem a está invocando.

Sua definição será:

Seção: Funções auxiliares Weaver (continuação):

```
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;
  <Seção a ser Inserida: Descobre tamanho do bloco do sistema de arquivos>
  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. Inicialização das Variáveis

6.1. inside_weaver_directory e project_path: Onde estamos

A primeira das variáveis é `inside_weaver_directory` , que deve valer `false` se o programa foi invocado de fora de um diretório de projeto Weaver e `true` caso contrário.

Como definir se estamos em um diretório que pertence à um projeto Weaver? Simples. São diretórios que contém dentro de si ou em um diretório ancestral um diretório oculto chamado `.weaver`. Caso encontremos este diretório oculto, também podemos aproveitar e ajustar a variável `project_path` para apontar para o local onde ele está. Se não o encontrarmos, estaremos fora de um diretório Weaver e não precisamos mudar nenhum valor das duas variáveis, pois elas deverão permanecer com o valor padrão `NULL` .

Em suma, o que precisamos é de um loop com as seguintes características:

Invariantes: A variável `complete_path` deve sempre possuir o caminho completo do diretório `.weaver` se ele existisse no diretório atual.

Inicialização: Inicializamos tanto o `complete_path` para serem válidos de acordo com o diretório em que o programa é invocado.

Manutenção: Em cada iteração do loop nós verificamos se encontramos uma condição de finalização. Caso contrário, subimos para o diretório pai do qual estamos, sempre atualizando as variáveis para que o invariante continue válido.

Finalização: Interrompemos a execução do loop se uma das três condições ocorrerem:

a) `complete_path == "/.weaver"` : Neste caso não podemos subir mais na árvore de diretórios, pois estamos na raiz do sistema de arquivos. Não encontramos um diretório `.weaver`. Isso significa que não estamos dentro de um projeto Weaver.

b) `complete_path == "C:\\.weaver"` : A letra inicial pode não ser um "C". De qualquer forma, estamos na raíz do sistema dos arquivos e não podemos subir mais como no caso acima. Com a diferença de estarmos no Windows.

c) `complete_path == "./.weaver"` e tal arquivo existe e é diretório: Neste caso encontramos um diretório `.weaver` e descobrimos que estamos dentro de um projeto Weaver. Podemos então atualizar a variável `project_path` para o diretório em que paramos.

O código de inicialização destas variáveis será então:

Seção: Inicialização:

```
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();
```

Para obtermos o diretório atual, vamos precisar do cabeçalho:

Seção: Cabeçalhos Incluídos no Programa Weaver:

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

Agora iniciamos um loop que terminará quando `complete_path` for igual à `/.weaver` (chegamos no fim da árvore de diretórios e não encontramos nada) ou quando realmente existir o diretório `.weaver/` no diretório examinado. E no fim do loop, sempre vamos para o diretório-pai do qual estamos:

Seção: Inicialização (continuação):

```
{
  size_t tmp_size = strlen(complete_path);
  // Testa se chegamos ao fim:
  while(strcmp(complete_path, "/.weaver") &&
               strcmp(complete_path, "\\.weaver") &&
               strcmp(complete_path + 1, ":\\.weaver")){
    if(directory_exist(complete_path) == EXISTE_E_EH_DIRETORIO){
      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__
      strlcat(complete_path, "/.weaver", tmp_size);
#else
      strcat(complete_path, "/.weaver");
#endif
    }
  }
  free(complete_path);
}
```

Como alocamos memória para `project_path` armazenar o endereço do projeto atual se estamos em um projeto Weaver, no final do programa teremos que desalocar a memória:

Seção: Finalização:

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

6.2. weaver_version_major e weaver_version_minor: Versão do Programa

Para descobrirmos a versão atual do Weaver que temos, basta consultar o valor presente na macro `VERSION`. Então, obtemos o número de versão maior e menor que estão separados por um ponto (se existirem). Note que se não houver um ponto no nome da versão, então ela é uma versão de testes. Mesmo neste caso o código abaixo vai funcionar, pois a função `atoi` iria retornar 0 nas duas invocações por encontrar respectivamente uma string sem dígito algum e um fim de string sem conteúdo:

Seção: Inicialização (continuação):

```
{
  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: Versão do Projeto

Se estamos dentro de um projeto Weaver, temos que inicializar informação sobre qual versão do Weaver foi usada para atualizá-lo pela última vez. Isso pode ser obtido lendo o arquivo *.weaver/version* localizado dentro do diretório Weaver. Se não estamos em um diretório Weaver, não precisamos inicializar tais valores. O número de versão maior e menor é separado por um ponto.

Seção: Inicialização (continuação):

```
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: Argumentos de Invocação

Uma das variáveis mais fáceis e triviais de se inicializar. Basta consultar `argc` e `argv`.

Seção: Inicialização (continuação):

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

6.5. arg_is_path: Se argumento é diretório

Agora temos que verificar se no caso de termos um argumento, se ele é um caminho para um projeto Weaver existente ou não. Para isso, checamos se ao concatenarmos `/.weaver` no argumento encontramos o caminho de um diretório existente ou não.

Seção: Inicialização (continuação):

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

6.6. shared_dir: Onde arquivos estão instalados

A variável `shared_dir` deverá conter onde estão os arquivos compartilhados da instalação de Weaver. Tais arquivos são as próprias bibliotecas a serem inseridas estaticamente e modelos de código fonte. Se existir a macro passada durante a compilação `WEAVER_DIR`, este será o caminho em que estão os arquivos. Caso contrário, assumiremos o valor padrão de `/usr/local/share/weaver` em sistemas baseados em Unix e o local apontado pela variável de ambiente ProgramFiles em ambientes Windows.

Seção: Inicialização (continuação):

```
{
#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();
}
```

Com isso damos poder durante a compilação para determinar onde os dados do motor Weaver serão armazenados no sistema. Algo mais comum de ser alterado em sistemas Unix que no Windows, onde espera-se que os programas sejam armazenados no mesmo lugar.

No Windows o código é mais longo principalmente por termos que determinar manualmente o nome do local padrão de se armazenar os programas. O endereço pode variar de acordo com o idioma do sistema, com a unidade de volume em que ele está ou com o fato do programa ter sido compilado em máquina com 32 ou 64 bits.

No fim do programa devemos desalocar a memória alocada para `shared_dir`:

Seção: Finalização (continuação):

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

6.7. arg_is_valid_project: Se o argumento é um nome de projeto

A próxima questão que deve ser averiguada é se o que recebemos como argumento, caso haja argumento, pode ser o nome de um projeto Weaver válido ou não. Para isso, três condições precisam ser satisfeitas:

1) O nome base do projeto deve ser formado somente por caracteres alfanuméricos e underline (embora uma barra possa aparecer para passar o caminho completo de um projeto).

2) Não pode existir um arquivo com o mesmo nome do projeto no local indicado para a criação.

3) O projeto não pode ter o nome de nenhum arquivo que costuma ficar no diretório base de um projeto Weaver (como "Makefile"). Do contrário, na hora da

compilação comandos como "`gcc game.c -o Makefile`" poderiam ser executados e sobrescreveriam arquivos importantes.

Para isso, usamos o seguinte código:

Seção: Inicialização (continuação):

```
if(have_arg && !arg_is_path){
  char *buffer;
  char *base = basename(argument);
  int size = strlen(base);
  int i;
  // Checando caracteres inválidos no nome:
  for(i = 0; i < size; i ++){
    if(!isalnum(base[i]) && base[i] != '_'){
      goto NOT_VALID;
    }
  }
  // Checando se arquivo existe:
  if(directory_exist(argument) != NAO_EXISTE){
    goto NOT_VALID;
  }
  // Checando se conflita com arquivos de compilação:
  buffer = concatenate(shared_dir, "project/", base, "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) != NAO_EXISTE){
    free(buffer);
    goto NOT_VALID;
  }
  free(buffer);
  arg_is_valid_project = true;
}
NOT_VALID:
```

Para podermos checar se um caractere é alfanumérico, incluimos a seguinte biblioteca:

Seção: Cabeçalhos Incluídos no Programa Weaver:

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

6.8. arg_is_valid_module: Se o argumento pode ser um nome de módulo

Checar se o argumento que recebemos pode ser um nome válido para um módulo só faz sentido se estivermos dentro de um diretório Weaver e se um argumento estiver sendo passado. Neste caso, o argumento é um nome válido se ele contiver apenas caracteres alfanuméricos, underline e se não existir no projeto um arquivo `.c` ou `.h` em `src/` que tenha o mesmo nome do argumento passado:

Seção: Inicialização (continuação):

```
if(have_arg && inside_weaver_directory){
  char *buffer;
  int i, size;
  size = strlen(argument);
  // Checando caracteres inválidos no nome:
  for(i = 0; i < size; i ++){
    if(!isalnum(argument[i]) && argument[i] != '_'){
      goto NOT_VALID_MODULE;
    }
```

```
  }
  // Checando por conflito de nomes:
  buffer = concatenate(project_path, "src/", argument, ".c", "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) != NAO_EXISTE){
    free(buffer);
    goto NOT_VALID_MODULE;
  }
  buffer[strlen(buffer) - 1] = 'h';
  if(directory_exist(buffer) != NAO_EXISTE){
    free(buffer);
    goto NOT_VALID_MODULE;
  }
  free(buffer);
  arg_is_valid_module = true;
}
NOT_VALID_MODULE:
```

6.9. arg_is_valid_plugin: Se o argumento pode ser um nome de plugin

Para que um argumento seja um nome válido para plugin, ele deve ser composto só por caracteres alfanuméricos ou underline e não existir no diretório `plugin` um arquivo com a extensão `.c` de mesmo nome. Também precisamos estar naturalmente, em um diretório Weaver.

Seção: Inicialização (continuação):

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

6.10. arg_is_valid_function: Se o argumento pode ser um nome de função de loop principal

Para que essa variável seja verdadeira, é preciso existir um segundo argumento e ele deve ser formado somente por caracteres alfanuméricos ou underline. Além disso, o primeiro caractere precisa ser uma letra e ele não pode ter o mesmo nome de alguma palavra reservada em C ou C++. A última versão checada para obter as

palavras reservadas foi o rascunho da especificação C++20 e o padrão C11.

Seção: Inicialização (continuação):

```
if(argument2 != NULL && inside_weaver_directory &&
   !strcmp(argument, "--loop")){
  int i, size;
  char *buffer;
  // Primeiro caractere não pode ser dígito
  if(isdigit(argument2[0]))
    goto NOT_VALID_FUNCTION;
  size = strlen(argument2);
  // Checando caracteres inválidos no nome:
  for(i = 0; i < size; i ++){
    if(!isalnum(argument2[i]) && argument2[i] != '_'){
      goto NOT_VALID_PLUGIN;
    }
  }
  // Checando se existem arquivos com o nome indicado:
  buffer = concatenate(project_path, "src/", argument2, ".c", "");
  if(buffer == NULL) W_ERROR();
  if(directory_exist(buffer) != NAO_EXISTE){
    free(buffer);
    goto NOT_VALID_FUNCTION;
  }
  buffer[strlen(buffer)-1] = 'h';
  if(directory_exist(buffer) != NAO_EXISTE){
    free(buffer);
    goto NOT_VALID_FUNCTION;
  }
  free(buffer);
  // Checando se recebemos como argumento uma palavra reservada em C/C++:
  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: Nome do criador do código

A variável `author_name` deve conter o nome do usuário que está invocando o programa. Esta informação é útil para gerar uma mensagem de Copyright nos arquivos de código fonte de novos módulos.

Isso será feito de maneira diferente em sistemas Unix e Windows. Em sistemas Unix, começamos obtendo o seu UID. De posse dele, obtemos todas as informações de login com um `getpwuid` . Se o usuário tiver registrado um nome em `/etc/passwd`, obtemos tal nome na estrutura retornada pela função. Caso contrário, assumiremos o login como sendo o nome:

Seção: Inicialização (continuação):

```
#if !defined(_WIN32)
{
  struct passwd *login;
  int size;
  char *string_to_copy;
  login = getpwuid(getuid()); // Obtém dados de usuário
  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
```

No Windows, o nome pode ser obtido com a função `GetUserNameExA` . Na primeira invocação tentamos obter o tamanho do buffer necessário para armazenarmos o nome e na segunda obtemos o nome em si. Em caso de erro, usamos a função mais antiga `GetUserNameA` que vai retornar um nome de usuário simples ao invés de tentar obter o nome completo, e para isso alocamos um espaço para o maior nome de usuário válido no sistema.

Seção: Inicialização (continuação):

```
#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
```

Depois, precisaremos desalocar a memória ocupada por `author_name` :

Seção: Finalização (continuação):

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

Para que o código funcione, devemos inserir uma biblioteca diferente dependendo de estarmos em sistemas Unix (para ter `getpwuid`) ou em sistemas Windows (para obtermos uma enumeração com diferentes formatos de nomes):

Seção: Cabeçalhos Incluídos no Programa Weaver:

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

6.12. project_name: Nome do projeto

Só faz sendido falarmos no nome do projeto se estivermos dentro de um projeto Weaver. Neste caso, o nome do projeto pode ser encontrado em um dos arquivos do diretório base de tal projeto em `.weaver/name`:

Seção: Inicialização (continuação):

```
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();
}
```

Depois, precisaremos desalocar a memória ocupada por `project_name`:

Seção: Finalização (continuação):

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

6.13. year: Ano atual

O ano atual é trivial de descobrir usando a função `localtime`, independente do sistema operacional:

Seção: Inicialização (continuação):

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

O único pré-requisito é incluirmos antes a biblioteca com funções de tempo:

Seção: Cabeçalhos Incluídos no Programa Weaver:

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

7. Casos de Uso

7.1. Imprimir ajuda de criação de projeto

O primeiro caso de uso sempre ocorre quando Weaver é invocado fora de um diretório de projeto e a invocação é sem argumentos ou com argumento `--help`. Nesse caso assumimos que o usuário não sabe bem como usar o programa e imprimimos uma mensagem de ajuda. A mensagem de ajuda terá uma forma semelhante a esta:

```
    .  .   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.
```

O que é feito com o código abaixo:

Seção: Caso de uso 1: Imprimir ajuda (criar projeto):

```
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. Imprimir ajuda de gerenciamento

O segundo caso de uso também é bastante simples. Ele é invocado quando já estamos dentro de um projeto Weaver e invocamos Weaver sem argumentos ou com um `--help`. Assumimos neste caso que o usuário quer instruções sobre a criação de um novo módulo. A mensagem que imprimiremos é semelhante à esta:

```
       \             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/
```

O que é obtido com o código:

Seção: Caso de uso 2: Imprimir ajuda de gerenciamento:

```
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. Mostrar a versão instalada de Weaver

Um caso de uso ainda mais simples. Ocorrerá toda vez que o usuário invocar Weaver com o argumento `--version`:

Seção: Caso de uso 3: Mostrar versão:

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

7.4. Atualizar projetos Weaver já existentes

Este caso de uso ocorre quando o usuário passar como argumento para Weaver um caminho absoluto ou relativo para um diretório Weaver existente. Assumimos então que ele deseja atualizar o projeto passado como argumento. Talvez o projeto tenha sido feito com uma versão muito antiga do motor e ele deseja que ele passe a usar uma versão mais nova da API.

Naturalmente, isso só será feito caso a versão de Weaver instalada seja igual ou superior à versão do projeto ou se a versão de Weaver instalada for uma versão instável para testes. Entende-se neste caso que o usuário deseja testar a versão experimental de Weaver no projeto. Fora isso, não é possível fazer *downgrades* de projetos, passando da versão 0.2 para 0.1, por exemplo.

Se a versão do projeto e a versão instalada forem as mesmas, nós ainda realizamos a atualização. Isso fornece um modo útil de corrigir um projeto se alguém apagar ou corromper um arquivo importante em `src/weaver/`.

Versões experimentais sempre são identificadas como tendo um nome formado somente por caracteres alfabéticos. Versões estáveis serão sempre formadas por um ou mais dígitos, um ponto e um ou mais dígitos (o número de versão maior e menor). Como o número de versão é interpretado com um `atoi` , isso significa que se estamos usando uma versão experimental, então o número de versão maior e menor serão sempre identificados como zero.

Projetos em versões experimentais de Weaver sempre serão atualizados, independente da versão ser mais antiga ou mais nova.

Uma atualização consiste em copiar todos os arquivos que estão no diretório de arquivos compartilhados Weaver dentro de `project/src/weaver` para o diretório `src/weaver` do projeto em questão. Para isso podemos contar com as funções de cópia de arquivos definidos na seção de funções auxiliares.

Seção: Caso de uso 4: Atualizar projeto Weaver:

```
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. Adicionando um módulo ao projeto Weaver

Se estamos dentro de um diretório de projeto Weaver, e o programa recebeu um argumento, então estamos inserindo um novo módulo no nosso jogo. Se o argumento é um nome válido, podemos fazer isso. Caso contrário,devemos imprimir uma mensagem de erro e sair.

Criar um módulo basicamente envolve:

a) Criar arquivos `.c` e `.h` base, deixando seus nomes iguais ao nome do módulo criado.

b) Adicionar em ambos um código com copyright e licenciamento com o nome do autor, do projeto e ano.

c) Adicionar no `.h` código de macro simples para evitar que o cabeçalho seja inserido mais de uma vez e fazer com que o `.c` inclua o `.h` dentro de si.

d) Fazer com que o `.h` gerado seja inserido em `src/includes.h` e assim suas estruturas sejam acessíveis de todos os outros módulos do jogo.

O código para isso é:

Seção: Caso de uso 5: Criar novo módulo:

```
if(inside_weaver_directory && have_arg &&
   strcmp(argument, "--plugin") && strcmp(argument, "--shader") &&
   strcmp(argument, "--loop")){
  if(arg_is_valid_module){
    char *filename;
    FILE *fp;
    // Criando modulo.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'; // Criando modulo.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);
```

```
    // Atualizando src/includes.h para inserir 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. Criar novo projeto

Criar um novo projeto Weaver consiste em criar um novo diretório com o nome do projeto, copiar para lá tudo o que está no diretório `project` do diretório de arquivos compartilhados e criar um diretório `.weaver` com os dados do projeto. Além disso, criamos um `src/game.c` e `src/game.h` adicionando o comentário de Copyright neles e copiando a estrutura básica dos arquivos do diretório compartilhado `basefile.c` e `basefile.h`. Também criamos um `src/includes.h` que por hora estará vazio, mas será modificado na criação de futuros módulos.

Seção: Caso de uso 6: Criar novo projeto:

```
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) //cd
    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); //Criando arquivo com número de versão:
```

```
    fp = fopen(".weaver/version", "w");
    fprintf(fp, "%s\n", VERSION);
    fclose(fp); // Criando arquivo com nome de projeto:
    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. Criar novo plugin

Este aso de uso é invocado quando temos dois argumentos, o primeiro é `"--plugin"` e o segundo é o nome de um novo plugin, o qual deve ser um nome único, sem conflitar com qualquer outro dentro de `plugins/`. Devemos estar em um diretório Weaver para fazer isso.

Seção: Caso de uso 7: Criar novo plugin:

```
if(inside_weaver_directory && have_arg && !strcmp(argument, "--plugin") &&
   arg_is_valid_plugin){
  char *buffer;
  FILE *fp;
  /* Criando o arquivo: */
  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. Criar novo shader

Este caso de uso é similar ao anterior, mas possui algumas diferenças. Todo shader será um novo arquivo no formato GLSL dentro do diretório `shaders`. E além disso, seu nome terá sempre o formato dado pela expressão regular `[0-9][0-9]*-.*`. O(s) dígito(s) na primeira parte do nome deve ser único para cada shader de um mesmo projeto. E os números representados por tais dígitos devem ser sempre sequenciais, começando no 1 e incrementando-o a cada novo shader.

Este caso de uso será invocado somente quando o nosso primeiro argumento for ‘‘`--shader`’’ e o segundo for um nome qualquer. Não precisamos realmente forçar uma restrição nos nomes dos shaders, pois sua convenção numérica garante que cada um terá um nome único e não-conflitante.

Para garantir isso, o código deverá contar quantos arquivos com extensão GLSL existem no diretório dos shaders e criar um novo shader com nome `DD-XX.glsl`, onde `DD` é o número de arquivos que existia mais 1 e `XX` é o nome escolhido passado como segundo argumento para o programa. Mas se existirem lacunas na numeração de shaders, por exemplo existir um shader 1 e um 3 sem existir o 2, daremos preferência para cobrir a lacuna.

Depois de descobrir a numeração do novo shader, basta criarmos ele como um arquivo vazio e depois copiarmos o conteúdo de um modelo já existente em nosso diretório de instalação.

O código deste caso de uso é então:

Seção: Caso de uso 8: Criar novo shader:

```
if(inside_weaver_directory && have_arg && !strcmp(argument, "--shader") &&
   argument2 != NULL){
    FILE *fp;
    size_t tmp_size, number = 0;
    int shader_number;
    char *buffer;
    <Seção a ser Inserida: Shader: Conta número de arquivos e obtém número
do shader>
    // Criando o 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();
}
```

A parte de contar onter o número do novo shader ocorre de maneira diferente no Unix e no Windows devido à API diferente para lidar com o sistema de arquivos. Tirando as particularidades de como iterar sobre arquivos em um diretório, o que faremos é iterar em cada arquivo no diretório `shaders` de nosso projeto que não seja um diretório, tenha extensão GLSL e tenha seu nome começado com um número positivo. Chamaremos tal número de `number`. Teremos um vetor booleano inicialmente marcado inteiramente como falso. Ao chegar em cada um destes arquivos, marcamos no vetor booleano a informação de que o shader de número `number` exis-

te colocando o valor verdadeiro na posição do vetor reservada para ele. Depois de iterarmos sobre cada um dos arquivos, acharemos a primeira posição do vetor que ainda está marcada como falsa. Sua posição indica qual número de shadr ainda não foi usado e é o número que escolheremos.

Complexidades adicionais neste código envolvem apenas tomarmos o cuidado para que o nosso vetor booleano sempre tenha um tamanho adequado. Para isso tentamos alocar ele inicialmente com 128 espaços, mas se acharmos shaders com números altos o bastante, o realocaremos para lidar com o número maior.

O código para isso no Linux será:

Seção: Shader: Conta número de arquivos e obtém número do shader:

```
#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) != EXISTE_E_EH_ARQUIVO){
      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){
      shader_number = i + 1;
      break;
    }
  free(exists);
}
#endif
```

No Windows, o código para iterar sobre arquivos é diferente, mas o restante não muda:

Seção: Shader: Conta número de arquivos e obtém número do shader:

```
#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) != EXISTE_E_EH_ARQUIVO){
```

```
      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){
    shader_number = i + 1;
    break;
  }
  free(exists);
}
#endif
```

7.9. Criar novo loop principal

Este caso de uso ocorre quando o segundo argumento é `--loop` e quando o próximo argumento for um nome válido para uma função. Se não for, imprimimos uma mensagem de erro para avisar.

Neste caso não podemos apenas copiar o conteúdo de um arquivo base para formar o arquivo com um novo módulo do projeto Weaver, pois esse novo arquivo terá definida uma função com um nome fornecido pelo usuário. Então apenas criamos e preenchemos o arquivo na hora com conteúdo defiido no próprio código abaixo.

Seção: Caso de uso 9: Criar novo loop principal:

```
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;
  // Criando 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);
  // Criando 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);
  // Atualizando src/includes.h
  fp = fopen("src/includes.h", "a");
  fprintf(fp, "#include \"%s.h\"\n", argument2);
  fclose(fp);
}
```

8. Conclusão

Isso finaliza todo o código necessário para que o programa Weaver possa gerenciar projetos de jogos feitos com o motor Weaver.

O programa apresentado aqui ainda não representa todo o gerenciamento de um projeto. Uma parte não retratada aqui é um instalador que em sistemas Unix é representado por um `Makefile` responsável por instalar o motor Weaver no local adequado e em sistemas Windows tem a forma de um pacote MSIX.

Além disso, o código das bibliotecas em si também fazem parte do motor Weaver, mas terão o seu código descrito em outros artigos.

Alguns códigos como o código-base para shaders e novos projetos podem ser encontrados junto com o código-fonte de Weaver, no diretório `project`.

Por fim, em sistemas Unix há um Makefile para cada projeto, que também realiza muito do desenvolvimento. No Windows, se utiliza-se o Visual Studio ao invés de ferramentas Unix, esta parte do gerenciamento será feita por um conjunto de regras.