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.*

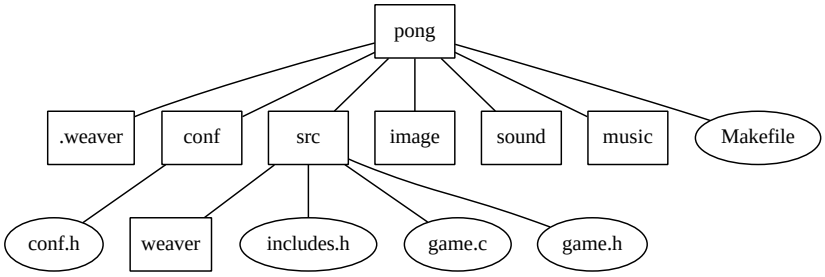
Resumo: *Este artigo descreve usando programação literária o programa Weaver. Este programa é um gerenciador de projetos para o Motor de Jogos Weaver. Se alguém deseja criar um novo projeto com o motor de jogos, usará este programa para criar a estrutura de diretórios desejada. Também usará o programa 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.

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.

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`", mas caso exista a variável de ambiente `WEAVER_DIR`, então este será considerado o endereço dos arquivos compartilhados.

`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 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

A última função auxiliar da qual precisaremos é uma função para concatenar strings. Ela deve receber um número arbitrário de srings como argumento, mas a última string deve ser uma string vazia. 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[0] != '\0'){ // Pára quando copiamos o ""
    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 caminho do diretório de arquivo

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:

```
#include <libgen.h>
```

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) ERROR();
complete_path = concatenate(path, "/.weaver", "");
free(path);
if(complete_path == NULL) 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") &&
```

```
Ψstrcmp(complete_path, "\.weaver") &&
Ψstrcmp(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); 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) ERROR();
  fp = fopen(file_path, "r");
  free(file_path);
  if(fp == NULL) ERROR();
  p = fgets(version, 10, fp);
  if(p == NULL){ fclose(fp); 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) 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):

```
{
  char *weaver_dir = NULL;
#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;
    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) 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) 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) 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) 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.

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) 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:
  if(!strcmp(argument2, "auto") || !strcmp(argument2, "break") ||
     !strcmp(argument2, "case") || !strcmp(argument2, "char") ||
     !strcmp(argument2, "const") || !strcmp(argument2, "continue") ||
     !strcmp(argument2, "default") || !strcmp(argument2, "do") ||
     !strcmp(argument2, "int") || !strcmp(argument2, "long") ||
     !strcmp(argument2, "register") || !strcmp(argument2, "return") ||
     !strcmp(argument2, "short") || !strcmp(argument2, "signed") ||
     !strcmp(argument2, "sizeof") || !strcmp(argument2, "static") ||
     !strcmp(argument2, "struct") || !strcmp(argument2, "switch") ||
     !strcmp(argument2, "typedef") || !strcmp(argument2, "union") ||
     !strcmp(argument2, "unsigned") || !strcmp(argument2, "void") ||
     !strcmp(argument2, "volatile") || !strcmp(argument2, "while") ||
     !strcmp(argument2, "double") || !strcmp(argument2, "else") ||
     !strcmp(argument2, "enum") || !strcmp(argument2, "extern") ||
     !strcmp(argument2, "float") || !strcmp(argument2, "for") ||
     !strcmp(argument2, "goto") || !strcmp(argument2, "if"))
    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) 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) 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, assumimos um tamanho padrão de 64 bytes e usamos a variável de ambiente `USERNAME`.

Seção: Inicialização (continuação):

```
#if defined(_WIN32)
{
  int size = 0;
  GetUserNameExA(NameDisplay, author_name, &size);
  if(size == 0)
    size = 64;
  author_name = (char *) malloc(size);
  if(GetUserNameExA(NameDisplay, author_name, &size) == 0){
    strncpy(author_name, getenv("USERNAME"), size);
    author_name[size - 1] = '\0';
  }
}
#endif
```

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

Seção: Finalização (continuação):

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

Para que o código Unix funcione, devemos inserir a biblioteca abaixo para termos acesso ao `getpwuid`:

Seção: Cabeçalhos Incluídos no Programa Weaver:

```
#if !defined(_WIN32)
#include <pwd.h> // getpwuid
#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) ERROR();
  project_name = (char *) malloc(256);
  if(project_name == NULL){
    free(filename);
    ERROR();
  }
  fp = fopen(filename, "r");
  if(fp == NULL){
    free(filename);
    ERROR();
  }
  c = fgets(project_name, 256, fp);
  fclose(fp);
  free(filename);
  if(c == NULL) ERROR();
  project_name[strlen(project_name)-1] = '\0';
  project_name = realloc(project_name, strlen(project_name) + 1);
  if(project_name == NULL) 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
```