# Capítulo 1: Introdução

Este é o código-fonte de `weaver`, uma *engine* (ou motor) para desenvolvimento de jogos feita em C utilizando-se da técnica de programação literária.

Um motor é um conjunto de bibliotecas e programas utilizado para facilitar e abstrair o desenvolvimento de um jogo. Jogos de computador, especialmente jogos em 3D são programas sofisticados demais e geralmente é inviável começar a desenvolver um jogo do zero. Um motor fornece uma série de funcionalidades genéricas que facilitam o desenvolvimento, tais como gerência de memória, renderização de gráficos bidimensionais e tridimensionais, um simulador de física, detector de colisão, suporte à animações, som, fontes, linguagem de script e muito mais.

Programação literária é uma técnica de desenvolvimento de programas de computador que determina que um programa deve ser especificado primariamente por meio de explicações didáticas de seu funcionamento. Desta forma, escrever um software que realiza determinada tarefa não deveria ser algo diferente de escrever um livro que explica didaticamente como resolver tal tarefa. Tal livro deveria apenas ter um rigor maior combinando explicações informais em prosa com explicações formais em código-fonte. Programas de computador podem então extrair a explicação presente nos arquivos para gerar um livro ou manual (no caso, este PDF) e também extrair apenas o código-fonte presente nele para construir o programa em si. A tarefa de montar o programa na ordem certa é de responsabilidade do programa que extrai o código. Um programa literário deve sempre apresentar as coisas em uma ordem acessível para humanos, não para máquinas.

Por exemplo, para produzir este PDF, utiliza-se um programa chamado TEX, o qual por meio do formato MAGITEX instalado, compreende código escrito em um formato específico de texto e o formata de maneira adequada. O TEX gera um arquivo no formato DVI, o qual é convertido para PDF. Para produzir o motor de desenvolvimento de jogos em si utiliza-se sobre os mesmos arquivos fonte um programa chamado CTANGLE, que extrai o código C (além de um punhado de códigos GLSL) para os arquivos certos. Em seguida, utiliza-se um compilador como GCC ou CLANG para produzir os executáveis. Felizmente, há `Makefiles` para ajudar a cuidar de tais detalhes de construção.

Os pré-requisitos para se compreender este material são ter uma boa base de programação em C e ter experiência no desenvolvimento de programas em C para Linux. Alguma noção do funcionamento de OpenGL também ajuda.

## 1.1 - Copyright e licenciamento

Weaver é desenvolvida pelo programador Thiago "Harry" Leucz Astrizi. Abaixo segue a licença do software:



A tradução não-oficial da licença é:

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

## 1.2 - Filosofia Weaver

Estes são os princípios filosóficos que guiam o desenvolvimento deste software. Qualquer coisa que vá de encontro à eles devem ser tratados como *bugs*.

**1- Software é conhecimento sobre como realizar algo escrito em linguagens formais de computadores. O conhecimento deve ser livre para todos. Portanto, Weaver deverá ser um software livre e deverá também ser usada para a criação de jogos livres.**

A arte de um jogo pode ter direitos de cópia. Ela deveria ter uma licença permisiva, pois arte é cultura, e portanto, também não deveria ser algo a ser tirado das pessoas. Mas weaver não tem como impedi-lo de licenciar a arte de um jogo da forma que for escolhida. Mas como Weaver funciona injetando estaticamente seu código em seu jogo e Weaver está sob a licença GPL, isso significa que seu jogo também deverá estar sob esta mesma licença (ou alguma outra compatível).

Basicamente isso significa que você pode fazer quase qualquer coisa que quiser com este software. Pode copiá-lo. Usar seu código-fonte para fazer qualquer coisa que queira (assumindo as responsabilidades). Passar para outras pessoas. Modificá-lo. A única coisa não permitida é produzir com ele algo que não dê aos seus usuários exatamente as mesmas liberdades.

As seguintes quatro liberdades devem estar presentes em Weaver e nos jogos que ele desenvolve:

Liberdade 0: A liberdade para executar o programa, para qualquer propósito.

Liberdade 1: A liberdade de estudar o software.

Liberdade 2: A liberdade de redistribuir cópias do programa de modo que você possa ajudar ao seu próximo.

Liberdade 3: A liberdade de modificar o programa e distribuir estas modificações, de modo que toda a comunidade se beneficie.

**2- Weaver deve estar bem-documentado.**

As quatro liberdades anteriores não são o suficiente para que as pessoas realmente possam estudar um software. Código ofuscado ou de difícil compreensão dificulta que as pessoas a exerçam. Weaver deve estar completamente documentada. Isso inclui explicação para todo o código-fonte que o projeto possui. O uso de MAGITEX e CWEB é um reflexo desta filosofia.

Algumas pessoas podem estranhar também que toda a documentação do código-fonte esteja em português. Estudei por anos demais em universidade pública e minha educação foi paga com dinheiro do povo brasileiro. Por isso acho que minhas contribuições devem ser pensadas sempre em como retribuir à isto. Por isso, o português brasileiro será o idioma principal na escrita deste software.

Infelizmente, isso tamém conflita com o meu desejo de que este projeto seja amplamente usado no mundo todo. Geralmente espera-se que código e documentação esteja em inglês. Para lidar

com isso, pretendo que a documentação on-line e guia de referência das funções esteja em inglês. Os nomes de funções e de variáveis estarão em inglês. Mas as explicações aqui serão em português.

Com isso tento conciliar as duas coisas, por mais difícil que isso seja.

**3- Weaver deve ter muitas opções de configuração para que possa atender à diferentes necessidades.**

É terrível quando você tem que lidar com abominações como:

**Arquivo:** `/tmp/dummy.c`**:**

```
CreateWindow("nome da classe", "nome da janela", WS_BORDER | WS_CAPTION |
             WS_MAXIMIZE, 20, 20, 800, 600, handle1, handle2, handle3, NULL);
```

Cada projeto deve ter um arquivo de configuração e muito da funcionalidade pode ser escolhida lá. Escolhas padrão sãs devem ser escolhidas e estar lá, de modo que um projeto funcione bem mesmo que seu autor não mude nada nas configurações. E concentrando configurações em um arquivo, retiramos complexidade das funções. As funções não precisam então receber mais de 10 argumentos diferentes e não é necessário também ficar encapsulando os 10 argumentos em um objeto de configuração, o qual é mais uma distração que solução para a complexidade.

Em todo projeto Weaver haverá um arquivo de configuração `conf/conf.h`, que modifica o funcionamento do motor. Como pode ser deduzido pela extensão do nome do arquivo, ele é basicamente um arquivo de cabeçalho C onde poderão ter vários `#define` s que modificarão o funcionamento de seu jogo.

**4- Weaver não deve tentar resolver problemas sem solução. Ao invés disso, é melhor propor um acordo mútuo entre usuários.**

Computadores tornam-se coisas complexas porque pessoas tentam resolver neles problemas insolúveis. É como tapar o sol com a peneira. Você na verdade consegue fazer isso. Junte um número suficientemente grande de peneiras, coloque uma sobre a outra e você consegue gerar uma sombra o quão escura se queira. Assim são os sistemas modernos que usamos nos computadores.

Como exemplo de tais tentativas de solucionar problemas insolúveis, temos a tentativa de fazer com que Sistemas Operacionais proprietários sejam seguros e livres de vírus, garantir privacidade, autenticação e segurança sobre HTTP e até mesmo coisas como o gerenciamento de memória. Pode-se resolver tais coisas apenas adicionando camadas e mais camadas de complexidade, e mesmo assim, não funcionará em realmente 100% dos casos.

Quando um problema não tem uma solução satisfatória, isso jamais deve ser escondido por meio de complexidades que tentam amenizar ou sufocar o problema. Ao invés disso, a limitação natural da tarefa deve ficar clara para o usuário, e deve-se trabalhar em algum tipo de comportamento que deve ser seguido pela engine e pelo usuário para que se possa lidar com o problema combinando os esforços de humanos e máquinas naquilo que cada um dos dois é melhor em fazer.

**5- Um jogo feito usando Weaver deve poder ser instalado em um computador simplesmente distribuindo-se um instalador, sem necessidade de ir atrás de dependências.**

Este é um exemplo de problema insolúvel mencionado anteriormente. Para isso a API Weaver é inserida estaticamente em cada projeto Weaver ao invés de ser na forma de bibliotecas compartilhadas. Mesmo assim ainda haverão dependências externas. Iremos então tentar minimizar elas e garantir que as duas maiores distribuições Linux no DistroWatch sejam capazes de rodar os jogos sem dependências adicionais além daquelas que já vem instaladas por padrão.

**6- Wever deve ser fácil de usar. Mais fácil que a maioria das ferramentas já existentes.**

Isso é obtido mantendo as funções o mais simples possíveis e fazendo-as funcionar seguindo padrões que são bons o bastante para a maioria dos casos. E caso um programador saiba o que está fazendo, ele deve poder configurar tais padrões sem problemas por meio do arquivo `conf/conf.h`.

Desta forma, uma função de inicialização poderia se chamar `Winit()` e não precisar de nenhum argumento. Coisas como gerenciar a projeção das imagens na tela devem ser transparentessem precisar de uma função específica após os objetos que compõe o ambiente serem definidos.

## 1.3 - Instalando Weaver

Para instalar Weaver em um computador, assumindo que você está fazendo isso à partir do

código-fonte, basta usar o comando `make` e `make install` (o segundo comando como *root*).

Atualmente, os seguintes programas são necessários para se compilar Weaver:

**ctangle** ou **notangle**: Extrai o código C dos arquivos de `cweb/`.

**clang** ou **gcc**: Um compilador C que gera executáveis à partir de código C.

**make:** Interpreta e executa comandos do Makefile.

Os dois primeiros programas podem vir em pacotes chamados de **cweb** ou **noweb**. Adicionalmente, os seguintes programas são necessários para se gerar a documentação:

**TEX e MAGITEX**: Usado para ler o código-fonte CWEB e gerar um arquivo DVI.

**dvipdf**: Usado para converter um arquivo `.dvi` em um `.pdf`.

**graphviz**: Gera representações gráficas de grafos.

Além disso, para que você possa efetivamente usar Weaver criando seus próprios projetos, você também poderá precisar de:

**emscripten:** Compila código C para Javascript e assim rodar em um navegador.

**opengl:** Permite gerar executáveis nativos com gráficos em 3D.

**xlib:** Permite gerar executáveis nativos gráficos.

**xxd:** Gera representação hexadecimal de arquivos. Insere o código dos shaders no programa. Por motivos obscuros, algumas distribuições trazem este último programa no mesmo pacote do **vim**.

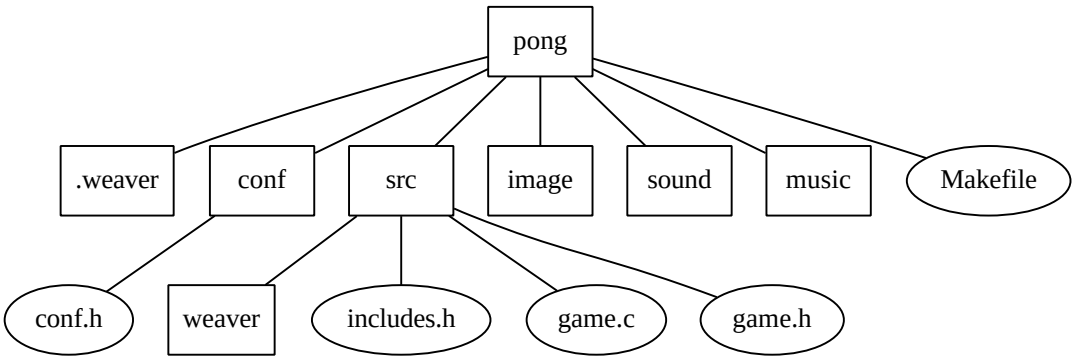
# 1.4 - O programa `weaver`

Weaver é uma engine para desenvolvimento de jogos que na verdaade é formada por várias coisas diferentes. Quando falamos em código do Weaver, podemos estar nos referindo à código de algum dos programas executáveis usados para se gerenciar a criação de seus jogos, podemos estar nos referindo ao código da API Weaver que é inserida em cada um de seus jogos ou então podemos estar nos referindo ao código de algum de seus jogos.

Para evitar ambigüidades, quando nos referimos ao programa executável, nos referiremos ao **programa Weaver**. Seu código-fonte será apresentado inteiramente neste capítulo. O programa é usado simplesmente para criar um novo projeto Weaver. E um projeto é um diretório com vários arquivos de desenvolvimento contendo código-fonte e multimídia. Por exemplo, o comando abaixo cria um novo projeto de um jogo chamado `pong`:

```
weaver pong
```

A árvore de diretórios exibida parcialmente abaixo é o que é criado pelo comando acima (diretórios são retângulos e arquivos são círculos):




Quando nos referimos ao código que é inserido em seus projetos, falamos do código da **API Weaver**. Seu código é sempre inserido dentro de cada projeto no diretório `src/weaver/`. Você terá aceso à uma cópia de seu código em cada novo jogo que criar, já que tal código é inserido estaticamente em seus projetos.

Já o código de jogos feitos com Weaver são tratados por **projetos Weaver**. É você quem escreve o seu código, ainda que a engine forneça como um ponto de partida o código inicial de inicialização, criação de uma janela e leitura de eventos do teclado e mouse.

## 1.4.1- Casos de Uso do Programa Weaver

Além de criar um projeto Weaver novo, o programa Weaver tem outros casos de uso. Eis a lista deles:

**Caso de Uso 1: Mostrar mensagem de ajuda de criação de novo projeto:** Isso deve ser feito toda vez que o usuário estiver fora do diretório de um Projeto Weaver e ele pedir ajuda explicitamente passando o parâmetro `--help` ou quando ele chama o programa sem argumentos (caso em que assumiremos que ele não sabe o que fazer e precisa de ajuda).

**Caso de Uso 2: Mostrar mensagem de ajuda do gerenciamento de projeto:** Isso deve ser feito quando o usuário estiver dentro de um projeto Weaver e pedir ajuda explicitamente com o argumento `--help` ou se invocar o programa sem argumentos (caso em que assumimos que ele não sabe o que está fazendo e precisa de ajuda).

**Caso de Uso 3: Mostrar a versão de Weaver instalada no sistema:** Isso deve ser feito toda vez que Weaver for invocada com o argumento `--version`.

**Caso de Uso 4: Atualizar um projeto Weaver existente:** Para o caso de um projeto ter sido criado com a versão 0.4 e tenha-se instalado no computador a versão 0.5, por exemplo. Para atualizar, basta passar como argumento o caminho absoluto ou relativo de um projeto Weaver. Independente de estarmos ou não dentro de um diretório de projeto Weaver. Atualizar um projeto significa mudar os arquivos com a API Weaver para que reflitam versões mais recentes.

**Caso de Uso 5: Criar novo módulo em projeto Weaver:** Para isso, devemos estar dentro do diretório de um projeto Weaver e devemos passar como argumento um nome para o módulo que não deve começar com pontos, traços, nem ter o mesmo nome de qualquer arquivo de extensão `.c` presente em `src/` (pois para um módulo de nome XXX, serão criados arquivos `src/XXX.c` e `src/XXX.h`).

**Caso de Uso 6: Criar um novo projeto Weaver:** Para isso ele deve estar fora de um diretório Weaver e deve passar como primeiro argumento um nome válido e não-reservado para seu novo projeto. Um nome válido deve ser qualquer um que não comece com ponto, nem traço, que não tenha efeitos negativos no terminal (tais como mudar a cor de fundo) e cujo nome não pode conflitar com qualquer arquivo necessário para o desenvolvimento (por exemplo, não deve-se poder criar um projeto chamado `Makefile`).

## 1.4.2- Variáveis 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

`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.

`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/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.

## 1.4.3- Estrutura Geral do Programa Weaver

Todas estas variáveis serão inicializadas no começo, e se precisar serão desalocadas no fim do programa, que terá a seguinte estrutura:

**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; /* Variáveis booleanas. */
  unsigned int project_version_major = 0, project_version_minor = 0,
    weaver_version_major = 0, weaver_version_minor = 0,
    year = 0;
  char *argument = NULL, *project_path = NULL, *shared_dir = NULL,
    *author_name = NULL, *project_name = NULL; /* Strings UTF-8 */

<Seção a ser Inserida: Inicialização>

  <Seção a ser Inserida: Caso de uso 1: Imprimir ajuda de criação de 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>

END_OF_PROGRAM:
<Seção a ser Inserida: Finalização>

  return return_value;
}
```

## 1.4.4- Macros 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;
```

## 1.4.5- Cabeçalhos do Programa Weaver

**Seção: Cabeçalhos Incluídos no Programa Weaver:**

```
#include <sys/types.h> // stat, getuid, getpwuid, mkdir
#include <sys/stat.h> // stat, mkdir
#include <stdbool.h> // bool, true, false
#include <unistd.h> // get_current_dir_name, getcwd, stat, chdir, getuid
#include <string.h> // strcmp, strcat, strcpy, strncmp
#include <stdlib.h> // free, exit, getenv
#include <dirent.h> // readdir, opendir, closedir
#include <libgen.h> // basename
#include <stdarg.h> // va_start, va_arg
#include <stdio.h> // printf, fprintf, fopen, fclose, fgets, fgetc, perror
#include <ctype.h> // isanum
#include <time.h> // localtime, time
#include <pwd.h> // getpwuid
```

## 1.4.6- Inicialização e Finalização do Programa Weaver

Inicializar Weaver significa inicializar as 14 variáveis que serão usadas para definir o seu comportamento.

### 1.4.6.1- Inicializando Variáveis `inside_weaver_directory` e `project_path`

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 duas 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 == ".weaver"` : 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.

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:

**Seção: Funções auxiliares Weaver:**

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

Note que caso a função receba uma string que não possua dois "/" em seu nome, obtemos um "buffer overflow" onde percorreríamos regiões de memória indevidas preenchendo-as com zero. Esta função é bastante perigosa, mas se limitarmos as strings que passamos para somente 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".

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:

**Seção: Funções auxiliares Weaver (continuação):**

```
int directory_exist(char *dir){
  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 0; // Não existe
  if(S_ISDIR(s.st_mode)) return 1; // Diretório
  return -1; // Arquivo
}
```

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;
  strcpy(new_string, string); // Copia primeira string

  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;
    strcat(new_string, current_string); // Copia próxima string
  }
  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 também é perigosa e deve ser usada sempre tomando-se este cuidado.

Por fim, podemos escrever agora o código de inicialização. Começamos primeiro fazendo `complete_path` ser igual à `./.weaver/`:

**Seção: Inicialização:**

```
char *path = NULL, *complete_path = NULL;
path = getcwd(NULL, 0);
if(path == NULL) ERROR();
complete_path = concatenate(path, "/.weaver", "");
free(path);
if(complete_path == NULL) ERROR();
```

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):**

```
while(strcmp(complete_path, "/.weaver")){ // Testa se chegamos ao fim
  if(directory_exist(complete_path) == 1){ // Testa se achamos o diretório
    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);
    strcat(complete_path, "/.weaver");
  }
}
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);
```

## 1.4.6.2- Inicializando variáveis `weaver_version_major` e `weaver_version_minor`

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);
}
```

## 1.4.6.3- Inicializando variáveis `project_version_major` e `project_version_minor`

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);
}
```

## 1.4.6.4- Inicializando `have_arg` e `argument`

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];
```

## 1.4.6.5- Inicializando `arg_is_path`

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) == 1){
    arg_is_path = 1;
  }
  free(buffer);
}
```

## 1.4.6.6- Inicializando `shared_dir`

A variável `shared_dir` deverá conter onde estão os arquivos compartilhados da instalação de Weaver. Se existir a variável de ambiente `WEAVER_DIR`, este será o caminho. Caso contrário, assumiremos o valor padrão de `/usr/share/weaver`.

**Seção: Inicialização (continuação):**

```
{
  char *weaver_dir = getenv("WEAVER_DIR");
  if(weaver_dir == NULL){
    shared_dir = concatenate("/usr/share/weaver/", "");
    if(shared_dir == NULL) ERROR();
  }
  else{
    shared_dir = concatenate(weaver_dir, "");
    if(shared_dir == NULL) ERROR();
  }
}
```

E isso requer que tenhamos que no fim do programa desalocar a memória alocada para `shared_dir`:

**Seção: Finalização (continuação):**

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

## 1.4.6.7- Inicializando `arg_is_valid_project`

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 (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])){
      goto NOT_VALID;
    }
  }
  // Checando se arquivo existe:
  if(directory_exist(argument) != 0){
    goto NOT_VALID;
  }
  // Checando se conflita com arquivos de compilação:
  buffer = concatenate(shared_dir, "project/", base, "");
  if(buffer == NULL) ERROR();
  if(directory_exist(buffer) != 0){
    free(buffer);
    goto NOT_VALID;
  }
  free(buffer);
  arg_is_valid_project = true;
}
NOT_VALID:
```

## 1.4.6.8- Inicializando `arg_is_valid_module`

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 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])){
      goto NOT_VALID_MODULE;
    }
  }
  // Checando por conflito de nomes:
  buffer = concatenate(project_path, "src/", argument, ".c", "");
  if(buffer == NULL) ERROR();
  if(directory_exist(buffer) != 0){
    free(buffer);
    goto NOT_VALID_MODULE;
  }
  buffer[strlen(buffer) - 1] = 'h';
  if(directory_exist(buffer) != 0){
    free(buffer);
    goto NOT_VALID_MODULE;
  }
  free(buffer);
  arg_is_valid_module = true;
}
NOT_VALID_MODULE:
```

## 1.4.6.9- Inicializando `author_name`

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.

Para obter o nome do usuário, 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):**

```
{
  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();
  strcpy(author_name, string_to_copy);
}
```

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

**Seção: Finalização (continuação):**

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

## 1.4.6.10- Inicializando `project_name`

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, *filename = concatenate(project_path, ".weaver/name", "");
  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);
```

## 1.4.6.11- Inicializando `year`

O ano atual é trivial de descobrir usando a função `localtime`:

**Seção: Inicialização (continuação):**

```
{
  time_t current_time;
  struct tm *date;

  time(&current_time);
  date = localtime(&current_time);
  year = date -> tm_year + 1900;
}
```

## 1.4.7- Caso de uso 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 de criação de 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();
}
```

## 1.4.8- Caso de uso 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
      /
```

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");
  END();
}
```

## 1.4.9- Caso de uso 3: Mostrar 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();
}
```

## 1.4.10- Caso de Uso 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 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.

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.

Pela definição que fizemos até agora, isso significa também que 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.

Mas para copiarmos os arquivos precisamos primeiro de uma função capaz de copiar um único arquivo. 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:

**Seção: Descobre tamanho do bloco do sistema de arquivos:**

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

De posse da função que copia um só arquivo, definimos uma função que copia todo o conteúdo de um diretório para outro diretório:

**Seção: Funções auxiliares Weaver (continuação):**

```
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)) mkdir(new_dst, 0755);
            if(copy_files(file, new_dst) == 0){
              free(new_dst);
              free(file);
              closedir(d);
              return 0; // Não fazemos nada para diretórios '.' e '..'
            }
          }
          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;
}
```

A função acima presumiu que o diretório de destino tem a mesma estrutura de diretórios que a origem.

De posse de todas as funções podemos escrever o código do caso de uso em que iremos realizar a atualização:

**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| passa a valer  SHARED_DIR/project/src/weaver
    buffer = concatenate(shared_dir, "project/src/weaver/", "");
    if(buffer == NULL) ERROR();
    // |buffer2| passa a valer PROJECT_DIR/src/weaver/
    buffer2 = concatenate(argument, "/src/weaver/", "");
    if(buffer2 == NULL){
      free(buffer);
      ERROR();
    }
    if(copy_files(buffer, buffer2) == 0){
      free(buffer);
      free(buffer2);
      ERROR();
    }
    free(buffer);
    free(buffer2);
  }
  END();
}
```

# 1.4.11- Caso de Uso 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.

A parte de imprimir um código de copyright será feita usando a nova função abaixo:

**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 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 General Public License for more\
  details.\n\nYou should have received a copy of the GNU General Public License\
 \nalong with %s. If not, see <http://www.gnu.org/licenses/>.*/\n\n";
  fprintf(fp, license, author_name, year, project_name, project_name,
          project_name, project_name);
}
```

Já o código de criação de novo módulo passa a ser:

**Seção: Caso de uso 5: Criar novo módulo:**

```
if(inside_weaver_directory && have_arg){
  if(arg_is_valid_module){
    char *filename;
    FILE *fp;
    // Criando modulo.c
    filename = concatenate(project_path, "src/", argument, ".c", "");
    if(filename == NULL) ERROR();
    fp = fopen(filename, "w");
    if(fp == NULL){
      free(filename);
      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);
      ERROR();
    }
    write_copyright(fp, author_name, project_name, year);
    fprintf(fp, "#ifndef _%s_h_\n", argument);
    fprintf(fp, "#define _%s_h_\n\n\n#endif", argument);
    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();
}
```

# 1.4.12- Caso de Uso 6: Criando um novo projeto Weaver

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.

A permissão dos diretórios criados será `drwxr-xr-x` ( `0755` em octal).

**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 = mkdir(argument, S_IRWXU | S_IRWXG | S_IROTH);
    if(err == -1) ERROR();
    err = chdir(argument);
    if(err == -1) ERROR();
    mkdir(".weaver", 0755); mkdir("conf", 0755);
    mkdir("src", 0755); mkdir("src/weaver", 0755);
    mkdir("image", 0755);  mkdir("sound", 0755);
    mkdir("music", 0755);

    dir_name = concatenate(shared_dir, "project", "");
    if(dir_name == NULL) ERROR();
    if(copy_files(dir_name, ".") == 0){
      free(dir_name);
      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) ERROR();
    write_copyright(fp, author_name, argument, year);
    if(append_basefile(fp, shared_dir, "basefile.c") == 0) ERROR();
    fclose(fp);

    fp = fopen("src/game.h", "w");
    if(fp == NULL) ERROR();
    write_copyright(fp, author_name, argument, year);
    if(append_basefile(fp, shared_dir, "basefile.h") == 0) ERROR();
    fclose(fp);
```

```
    fp = fopen("src/includes.h", "w");
    write_copyright(fp, author_name, argument, year);
    fclose(fp);
  }
  else{
    fprintf(stderr, "ERROR: %s is not a valid project name.", argument);
    return_value = 1;
  }
  END();
}
```

A única coisa ainda não-definida é a função usada acima `append_basefile`. Esta é uma função bastante específica para concatenar o conteúdo de um arquivo para o outro dentro deste trecho de código. Não é uma função geral, pois ela recebe como argumento um ponteiro para o arquivo de destino aberto e reebe como argumento o diretório em que está a origem e o nome do arquivo de origem ao invés de ter a forma mais intuitiva `cat(origem, destino)`.

Definimos abaixo a forma da `append_basefile`:

**Seção: Funções auxiliares Weaver (continuação):**

```
int append_basefile(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;
}
```

E isso conclui todo o código do Programa Weaver. Todo o resto de código que será apresentado à seguir, não pertence mais ao programa Weaver, mas à Projetos Weaver e à API Weaver.

# 1.5 - O arquivo `conf.h`

Em toda árvore de diretórios de um projeto Weaver, deve existir um arquivo cabeçalho C chamado`conf/conf.h`. Este cabeçalho será incluído em todos os outros arquivos de código do Weaver no projeto e que permitirá que o comportamento da Engine seja modificado naquele projeto específico.

O arquivo deverá ter as seguintes macros (dentre outras):

- `W_DEBUG_LEVEL` : Indica o que deve ser impresso na saída padrão durante a execução. Seu valor pode ser:
  - `0` ) Nenhuma mensagem de depuração é impressa durante a execução do programa. Ideal para compilar a versão final de seu jogo.
  - `1` ) Mensagens de aviso que provavelmente indicam erros são impressas durante a execução. Por exemplo, um vazamento de memória foi detectado, um arquivo de textura não foi encontrado, etc.
  - `2` ) Mensagens que talvez possam indicar erros ou problemas, mas que talvez sejam inofensivas são impressas.
  - `3` ) Mensagens informativas com dados sobre a execução, mas que não representam problemas são impressas.
  - `4` ) Código de teste adicional é executado apenas para garantir que condições que tornem o código incorreto não estão presentes. Use só se você está depurando ou desenvolvendo a própria API Weaver, não o projeto de um jogo que a usa.
- `W_SOURCE` : Indica a linguagem que usaremos em nosso projeto. As opções são:
  - `W_C` ) Nosso projeto é um programa em C.
  - `W_CPP` ) Nosso projeto é um programa em C++.
- `W_TARGET` : Indica que tipo de formato deve ter o jogo de saída. As opções são:
  - `W_ELF` ) O jogo deverá rodar nativamente em Linux. Após a compilação, deverá ser criado um arquivo executável que poderá ser instalado com `make install`.
  - `W_WEB` ) O jogo deverá executar em um navegador de Internet. Após a compilação deverá ser criado um diretório chamado `web` que conterá o jogo na forma de uma página HTML com Javascript. Não faz sentido instalar um jogo assim. Ele deverá ser copiado para algum servidor Web para que possa ser jogado na Internet. Isso é feito usando Emscripten.

Opcionalmente as seguintes macros podem ser definidas também (dentre outras):

- `W_MULTITHREAD` : Se a macro for definida, Weaver é compilado com suporte à múltiplas threads acionadas pelo usuário. Note que de qualquer forma vai existir mais de uma thread rodando no programa para que música e efeitos sonoros sejam tocados. Mas esta macro garante que mutexes e código adicional sejam executados para que o desenvolvedor possa executar qualquer função da API concorrentemente.

Ao longo das demais seções deste documento, outras macros que devem estar presentes ou que são opcionais serão apresentadas. Mudar os seus valores, adicionar ou removê-las é a forma de configurar o funcionamento do Weaver.

Junto ao código-fonte de Weaver deve vir também um arquivo `conf/conf.h` que apresenta todas as macros possíveis em um só lugar. Apesar de ser formado por código C, tal arquivo não será apresentado neste PDF, pois é importante que ele tenha comentários e CWEB iria remover os comentários ao gerar o código C.

O modo pelo qual este arquivo é inserido em todos os outros cabeçalhos de arquivos da API Weaver é:

**Seção: Inclui Cabeçalho de Configuração:**

```
#include "conf_begin.h"
#include "../../conf/conf.h"
```

Note que haverão também cabeçalhos `conf_begin.h` que cuidarão de toda declaração de inicialização que forem necessárias. Para começar, criaremos o `conf_begin.h` para inicializar as macros `W_WEB` e `W_ELF` :

**Arquivo:** `project/src/weaver/conf_begin.h`**:**

```
#define W_ELF 0
```

```
#define W_WEB 1
```

## 1.6 - Funções básicas Weaver

E agora começaremos a definir o começo do código para a API Weaver.

Primeiro criamos um `weaver.h` que irá incluir automaticamente todos os cabeçalhos Weaver necessários:

**Arquivo:** `project/src/weaver/weaver.h`:

```
#ifndef _weaver_h_
#define _weaver_h_
#ifdef __cplusplus
  extern "C" {
#endif
```

<Seção a ser Inserida: Inclui Cabeçalho de Configuração>

<Seção a ser Inserida: Cabeçalhos Weaver>

```
#ifdef __cplusplus
  }
#endif
#endif
```

Neste cabeçalho, iremos também declarar três funções.

A primeira função servirá para inicializar a API Weaver. Seus parâmetros devem ser o nome do arquivo em que ela é invocada e o número de linha. Esta informação será útil para imprimir mensagens de erro úteis em caso de erro.

A segunda função deve ser a última coisa invocada no programa. Ela encerra a API Weaver.

E a terceira função deve ser chamada no loop principal do programa e será responsável pro fazer coisas como desenhar na tela, ficar um tempo ociosa para não consumir 100% da CPU e coisas assim. O seu argumento representa quantos milissegundos devemos ficar nela sem fazer nada para evitar consumo de todo o tempo de CPU.

Nenhuma destas funções foi feita para ser chamada por mais de uma thread. Todas elas só devem ser usadas pela thread principal. Mesmo que você defina a macro `W_MULTITHREAD`, todas as outras funções serão seguras para threads, menos estas três.

**Seção: Cabeçalhos Weaver (continuação):**

```
void _awake_the_weaver(void);
void _may_the_weaver_sleep();
void _weaver_rest(unsigned long time);

#define Winit() _awake_the_weaver()
#define Wexit() _may_the_weaver_sleep()
#define Wrest(a) _weaver_rest(a)
```

Definiremos melhor a responsabilidade destas funções ao longo dos demais capítulos. Mas colocaremos aqui a definição delas no arquivo adequado. E no caso da função `_weaver_rest`, colocaremos um pouco de seu código.

A função `_weaver_rest` é a função a ser executada em cada frame do jogo. Ela executa de forma diferente se o programa está sendo compilado para um executável Linux ou para uma página de Internet via Emscripten.

Para dar uma pequena amostra do que ela faz, segue um código para ela em que a função limpa os buffers OpenGL ( `glClear` ), executa todo o código relevante ao loop principal do jogo (que iremos definir em breve), troca os buffers de desenho na tela ( `glXSwapBuffers` , somente se

formos um programa executável, não algo compilado para Javascript), pede que todos os comandos OpenGL pendentes sejam executados ( `glFlush` ) e, se pertinente, pede que o programa fique um tempo ocioso para não usar 100% da CPU ( `nanosleep` , só para programa executável, não se compilado para Javascript).

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

```
#include "weaver.h"

                    <Seção a ser Inserida: API Weaver: Definições>

void _awake_the_weaver(void){
                    <Seção a ser Inserida: API Weaver: Inicialização>
}

void _may_the_weaver_sleep(void){
                    <Seção a ser Inserida: API Weaver: Finalização>
  exit(0);
}

void _weaver_rest(unsigned long time){
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
#if W_TARGET == W_ELF
  struct timespec req = {0, time * 1000000};
#endif
                  <Seção a ser Inserida: API Weaver: Loop Principal>
#if W_TARGET == W_ELF
  glXSwapBuffers(_dpy, _window);
#else
  glFlush();
#endif
#if W_TARGET == W_ELF
  nanosleep(&req, NULL);
#endif
}
```

Mas isso é só uma amostra inicial e uma inicialização dos arquivos. Estas funções todas serão mais ricamente definidas a cada capítulo à medida que definimos novas responsabilidades para o nosso motor de jogo.

**Seção: API Weaver: Loop Principal:**

```
  // A definir...
```

**Seção: API Weaver: Definições:**

```
  // A definir...
```

**Seção: API Weaver: Inicialização:**

```
  // A definir...
```

**Seção: API Weaver: Finalização:**

```
  // A definir...
```

## 1.7 - A estrutura `W`

As três funções que definimos acima são atípicas. A maioria das variáveis e funções que criaremos ao longo do projeto não serão definidas globalmente, mas serão atribuídas à uma estrutura.

Na prática estamos aplicando técnica de orientação à objetos, criando o Objeto “Weaver API” e definindo seus próprios atributos e métodos ao invés de termos que definir variáveis globais.

O objeto terá a forma:

**Seção: Cabeçalhos Weaver:**

```
// Esta estrutura conterá todas as variáveis e funções definidas pela
// API Weaver:
extern struct _weaver_struct W;
```

**Seção: API Weaver: Definições:**

```
struct _weaver_struct{
                    <Seção a ser Inserida: Variáveis Weaver>
                    <Seção a ser Inserida: Funções Weaver>
} W;
```

A vantagem de fazermos isso é evitarmos a poluição do espaço de nomes. Fazendo isso diminuimos muito a chance de existir algum conflito entre o nome que damos a uma variável global e um nome exportado por alguma biblioteca. As únicas funções com as quais não nos preocuparemos serão aquelas que começam com um “ _ ”, pois elas serão internas à API. Nenhum usuário deve criar funções que começam com o “ _ ”.

Uma vantagem ainda maior de fazermos isso é que passamos a ser capazes de passar a estrutura `W` para *plugins*, que normalmente não teriam como acessar coisas que estão como variáveis globais. Mas os *plugins* podem definir funções que recebem como argumento `W` e assim eles podem ler informações e manipular a API.

# 1.8 - Sumário das Variáveis e Funções da Introdução

Terminaremos todo capítulo deste livro/programa com um sumário de todas as funções e variáveis definidas ao longo do capítulo que estejam disponíveis na API Weaver. As funções do programa Weaver, bem como variáveis e funções estáticas serão omitidas. O sumário conterá uma descrição rápida e poderá ter algum código adicional que possa ser necessário para inicializá-lo e defini-lo.

•Este capítulo apresentou 1 nova variável da API Weaver:

`W` : Uma estrutura que irá armazenar todas as variáveis globais da API Weaver, bem como as suas funções globais. Exceto as três outras funções definidas neste capítulo.

•Este capítulo apresentou 3 novas funções da API Weaver:

`void Winit(void)` : Inicializa a API Weaver. Deve ser a primeira função invocada pelo programa antes de usar qualquer coisa da API Weaver.

`void Wexit(void)` : Finaliza a API Weaver. Deve ser chamada antes de encerrar o programa.

`void Wrest(unsigned long time)` : Deve ser invocada em cada iteração do *loop* principal do programa. O argumento especifica quantos milissegundos o programa deve ficar ocioso, liberando assim parte da CPU para o Sistema Operacional.

# Capítulo 2: Gerenciamento de memória

Alocar memória dinamicamente é uma operação cujo tempo nem sempre pode ser previsto. Depende da quantidade de blocos contínuos de memória presentes na heap que o gerenciador organiza. E isso depende muito do padrão de uso das funções `malloc` e `free`.

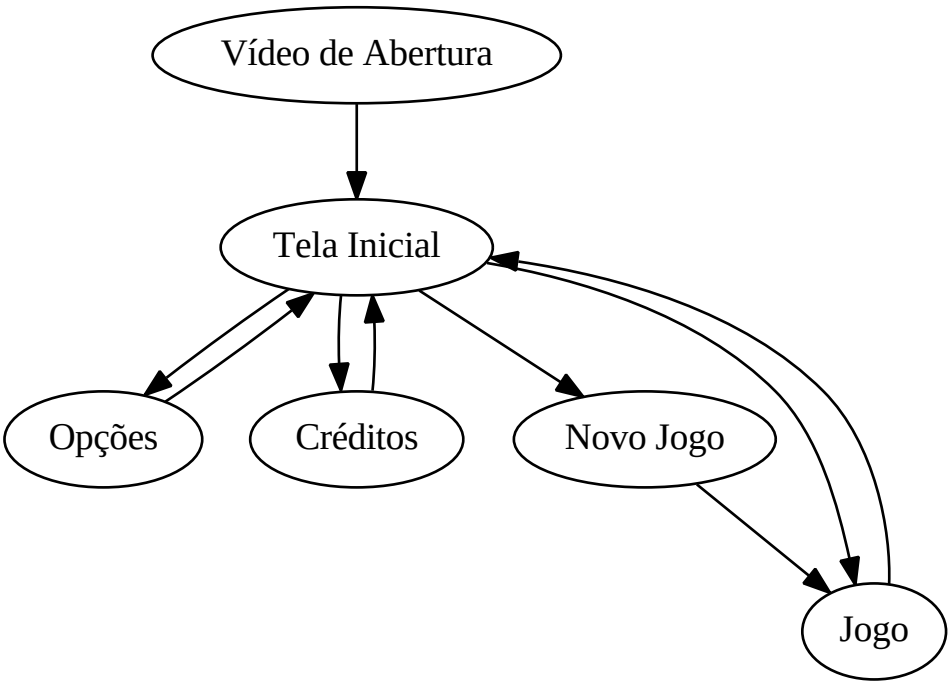
Jogos de computador tradicionalmente evitam o uso contínuo de `malloc` e `free` por causa disso. Tipicamente jogos programados para ter um alto desempenho alocam toda (ou a maior parte) da memória de que vão precisar logo no início da execução gerando um *pool* de memória e gerenciando ele ao longo da execução. De fato, esta preocupação direta com a memória é o principal motivo de linguagens sem *garbage collectors* como C++ serem tão preferidas no desenvolvimento de grandes jogos comerciais.

Um dos motivos para isso é que também nem sempre o `malloc` disponível pela biblioteca padrão de algum sistema é muito eficiente para o que está sendo feito. Como um exemplo, será mostrado posteriormente gráficos de benchmarks que mostram que após ser compilado para Javascript usando Emscripten, a função `malloc` da biblioteca padrão do Linux torna-se terrivelmente lenta. Mas mesmo que não estejamos lidando com uma implementação rápida, ainda assim há benefícios em ter um alocador de memória próprio. Pelo menos a práica de alocar toda a memória necessária logo no começo e depois gerenciar ela ajuda a termos um programa mais rápido.

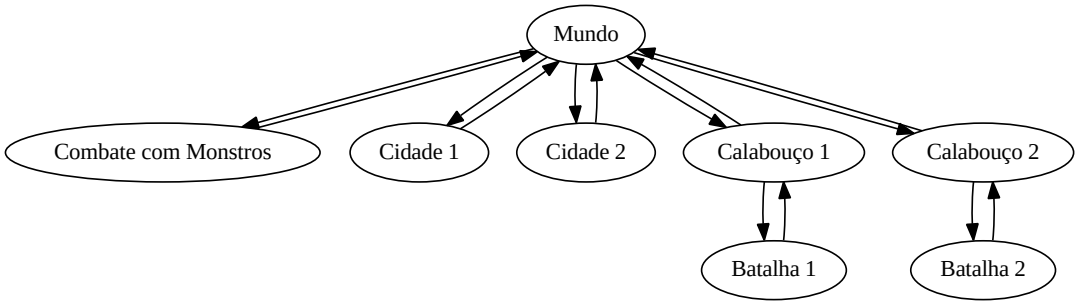
Por causa disso, Weaver exisge que você informe anteriormente quanto de memória você irá usar e cuida de toda a alocação durante a inicialização. Sabendo a quantidade máxima de memória que você vai usar, isso também permite que vazamentos de memória sejam detectados mais cedo e permitem garantir que o seu jogo está dentro dos requisitos de memória esperados.

Weaver de fato aloca mais de uma região contínua de memória onde pode-se alocar coisas. Uma das regiões contínuas será alocada e usada pela própria API Weaver à medida que for necessário. A segunda região de memória contínua, cujo tamanho deve ser declarada em `conf/conf.h` é a região dedicada para que o usuário possa alocar por meio de `Walloc` (que funciona como o `malloc`). Além disso, o usuário deve poder criar novas regiões contínuas de memória dentro das quais pode-se fazer novas alocações. O nome que tais regiões recebem é **arena**.

Além de um `Walloc`, também existe um `Wfree`. Entretanto, o jeito recomendável de desalocar na maioria das vezes é usando uma outra função chamada `Wtrash`. Para explicar a ideia de seu funcionamento, repare que tipicamente um jogo funciona como uma máquina de estados onde mudamos várias vezes de estado. Por exemplo, em um jogo de RPG clássico como Final Fantasy, podemos encontrar os seguintes estados:

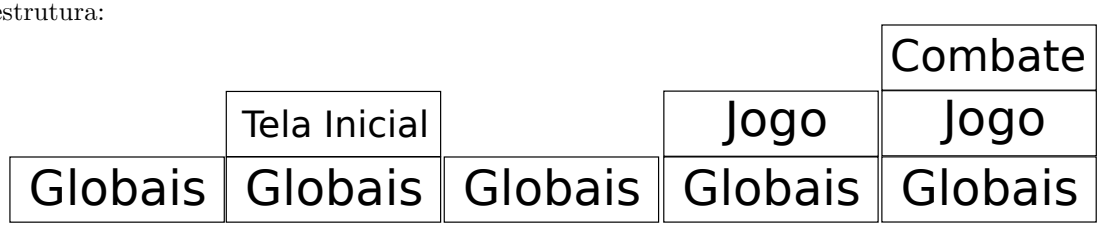


E cada um dos estados pode também ter os seus próprios sub-estados. Por exemplo, o estado "Jogo" seria formado pela seguinte máquina de estados interna:




Cada estado precisará fazer as suas próprias alocações de memória. Algumas vezes, ao passar de um estado pro outro, não precisamos lembrar do quê havia no estado anterior. Por exemplo, quando passamos da tela inicial para o jogo em si, não precisamos mais manter na memória a imagem de fundo da tela inicial. Outras vezes, podemos precisar memorizar coisas. Se estamos andando pelo mundo e somos atacados por monstros, passamos para o estado de combate. Mas uma vez que os monstros sejam derrotados, devemos voltar ao estado anterior, sem esquecer de informações como as coordenadas em que estávamos. Mas quando formos esquecer um estado, iremos querer sempre desalocar toda a memória relacionada à ele.

Por causa disso, um jogo pode ter um gerenciador de memória que funcione como uma pilha. Primeiro alocamos dados globais que serão úteis ao longo de todo o jogo. Todos estes dados só serão desalocados ao término do jogo. Em seguida, podemos criar um **breakpoint** e alocamos todos os dados referentes à tela inicial. Quando passarmos da tela inicial para o jogo em si, podemos desalocar de uma vez tudo o que foi alocado desde o último *breakpoint* e removê-lo. Ao entrar no jogo em si, criamos um novo *breakpoint* e alocamos tudo o que precisamos. Se entramos em tela de combate, criamos outro *breakpoint* (sem desalocar nada e sem remover o *breakpoint* anterior) e alocamos os dados referentes à batalha. Depois que ela termina, desalocamos tudo até o último *breakpoint* para apagarmos os dados relacionados ao combate e voltamos assim ao estado anterior de caminhar pelo mundo. Ao longo destes passos, nossa memória terá aproximadamente a seguinte estrutura:




Sendo assim, nosso gerenciador de memória torna-se capaz de evitar completamente fragmentação tratando a memória alocada na heap como uma pilha. O desenvolvedor só precisa desalocar a memória na ordem inversa da alocação (se não o fizer, então haverá fragmentação). Entretanto, a desalocação pode ser um processo totalmente automatizado. Toda vez que encerramos um estado, podemos ter uma função que desaloca tudo o que foi alocado até o último *breakpoint* na ordem correta e elimina aquele *breakpoint* (exceto o último na base da pilha que não pode ser eliminado). Fazendo isso, o gerenciamento de memória fica mais simples de ser usado, pois o próprio gerenciador poderá desalocar tudo que for necessário, sem esquecer e sem deixar vazamentos de memória. O que a função `Wtrash` faz então é desalocar na ordem certa toda a memória alocada até o último *breakpoint* e destrói o *breakpoint* (exceto o primeiro que nunca é removido). Para criar um novo *breakpoint*, usamos a função `Wbreakpoint` .

Tudo isso sempre é feito na arena padrão. Mas pode-se criar uma nova arena ( `Wcreate_arena` ) bem como destruir uma arena ( `Wdestroy_arena` ). E pode-se então alocar memória na arena personalizada criada ( `Walloc_arena` ) e desalocar ( `Wfree_arena` ). Da mesmo forma, pode-se também criar um *breakpoint* na arena personalizada ( `Wbreakpoint_arena` ) e descartar tudo que foi alocado nela até o último *breakpoint* ( `Wtrash_arena` ).

Para garantir a inclusão da definição de todas estas funções e estruturas, usamos o seguinte código:

**Seção: Cabeçalhos Weaver:**

```
#include "memory.h"
```

E também criamos o cabeçalho de memória. À partir de agora, cada novo módulo de Weaver terá um nome associado à ele. O deste é "Memória". E todo cabeçalho `.h` dele conterá, além das macros comuns para impedir que ele seja inserido mais de uma vez e para que ele possa ser usado em C++, uma parte na qual será inserido o cabeçalho de configuração (visto no fim do capítulo anterior) e a parte de declarações, com o nome `Declarações de NOME_DO_MODULO`.

**Arquivo:** `project/src/weaver/memory.h`**:**

```
#ifndef _memory_h_
#define _memory_h_
#ifdef __cplusplus
  extern "C" {
#endif
            <Seção a ser Inserida: Inclui Cabeçalho de Configuração>
                <Seção a ser Inserida: Declarações de Memória>
#ifdef __cplusplus
  }
#endif
#endif
```

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

```
#include "memory.h"
```

No caso, as Declarações de Memória que usaremos aqui começam com os cabeçalhos que serão usados, e posteriormente passarão para as declarações das funções e estruturas de dado a serem usadas nele:

**Seção: Declarações de Memória:**

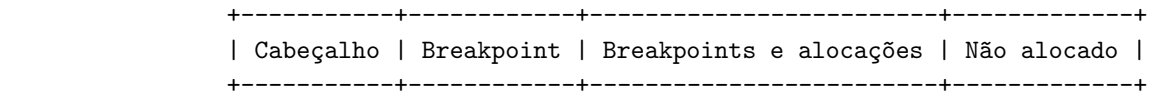
```
#include <sys/mman.h> // |mmap|, |munmap|
#include <pthread.h> // |pthread_mutex_init|, |pthread_mutex_destroy|
#include <string.h> // |strncpy|
#include <unistd.h> // |sysconf|
#include <stdlib.h> // |size_t|
#include <stdio.h> // |perror|
#include <math.h> // |ceil|
#include <stdbool.h>
```

Outra coisa relevante a mencionar é que à partir de agora assumiremos que as seguintes macros são definidas em `conf/conf.h`:

- `W_MAX_MEMORY` : O valor máximo em bytes de memória que iremos alocar por meio da função `Walloc` de alocação de memória na arena padrão.
- `W_WEB_MEMORY` : A quantidade de memória adicional em bytes que reservaremos para uso caso compilemos o nosso jogo para a Web ao invés de gerar um programa executável. O Emscripten precisará de memória adicional e a quantidade pode depender do quanto outras funções como `malloc` e `Walloc_arena` são usadas. Este valor deve ser aumentado se forem encontrados problemas de falta de memória na web. Esta macro será consultada na verdade por um dos `Makefiles`, não por código que definiremos neste PDF.

## 2.1 - Estruturas de Dados Usadas

Vamos considerar primeiro uma **arena**. Toda **arena** terá a seguinte estrutura:




A terceira região é onde toda a ação de alocação e liberação de memória ocorrerá. No começo estará vazia e a área não-alocada será a maioria. À medida que alocações e desalocações ocorrerem, a região de alocação e *breakpoints* crescerá e diminuirá, sempre substituindo o espaço não-alocado ao crescer. O cabeçalho e *breakpoint* inicial sempre existirão e não poderão ser removidos. O primeiro *breakpoint* é útil para que o comando `Wtrash` sempre funcione e seja definido, pois sempre existirá um último *breakpoint*.

A memória pode ser vista de três formas diferentes:

1) Como uma pilha que cresce da última alocação até a região não-alocada. Sempre que uma nova alocação é feita, ela será colocada imediatamente após a última alocação feita. Se memória for desalocada, caso a memória em questão esteja no fim da pilha, ela será efetivamente liberada. Caso contrário, será marcada para ser removida depois, o que infelizmente pode gerar fragmentação se o usuário não tomar cuidado.

2) Como uma lista duplamente encadeada. Cada *breakpoint* e região alocada terá ponteiros para a próxima região e para a região anterior (ou para `NULL`). Desta forma, pode-se percorrer rapidamente em uma iteração todos os elementos da memória.

3) Como uma árvore. Cada elemento terá um ponteiro para o último *breakpoint*. Desta forma, caso queiramos descartar a memória alocada até encontrarmos o último *breakpoint*, podemos consultar este ponteiro.

## 2.1.1- Cabeçalho da Arena

O cabeçalho conterá todas as informações que precisamos para usar a arena. Chamaremos sua estrutura de dados de `struct arena_header`.

O tamanho total da arena nunca muda. O cabeçalho e primeiro breakpoint também tem tamanho constante. A região de breakpoint e alocações pode crescer e diminuir, mas isso sempre implica que a região não-alocada respectivamente diminui e cresce na mesma proporção.

As informações encontradas no cabeçalho são:

- **Total:** A quantidade total em bytes de memória que a arena possui. Como precisamos garantir que ele tenha um tamanho suficientemente grande para que alcance qualquer posição que possa ser alcançada por um endereço, ele precisa ser um `size_t`. Pelo padrão ISO isso será no mínimo 2 bytes, mas em computadores pessoais atualmente está chegando a 8 bytes.

  Esta informação será preenchida na inicialização da arena e nunca mais será mudada.
- **Usado:** A quantidade de memória que já está em uso nesta arena. Isso nos permite verificar se temos espaço disponível ou não para cada alocação. Pelo mesmo motivo do anterior, precisa ser um `size_t`. Esta informação precisará ser atualizada toda vez que mais memória for alocada ou desalocada. Ou quando um *breakpoint* for criado ou destruído.
- **Último Breakpoint:** Armazenar isso nos permite saber à partir de qual posição podemos começar a desalocar memória em caso de um `Wtrash`. Outro `size_t`. Eta informação precisa ser atualizada toda vez que um *breakpoint* for criado ou destruído. Um último breakpoint sempre existirá, pois o primeiro breakpoint nunca pode ser removido.
- **Último Elemento:** Endereço do último elemento que foi armazenado. É útil guardar esta informação porque quando criamos um novo elemento com `Walloc` ou `Wbreakpoint`, o novo elemento precisa apontar para o último que havia antes dele. Esta informação precisa ser atualizada após qualquer operação de alocação, desalocação ou *breakpoint*. Sempre existirá um último elemento na arena, pois se nada foi alocado um primeiro breakpoint sempre estará posicionado após o cabeçalho e este será nosso último elemento.
- **Posição Vazia:** Um ponteiro para a próxima região contínua de memória não-alocada. É preciso saber disso para podermos criar novas estruturas e retornar um espaço ainda não-utilizado em caso de `Walloc`. Outro `size_t`. Novamente é algo que precisa ser atualizado após qualquer uma das operações de memória sobre a arena. É possível que não hajam mais regiões vazias caso tudo já tenha sido alocado. Neste caso, o ponteiro deverá ser `NULL`.

- **Mutex:** Opcional. Só precisamos definir isso se estivermos usando mais de uma thread. Neste caso, o mutex servirá para prevenir que duas threads tentem modificar qualquer um destes valores ao mesmo tempo. Caso seja usado, o mutex precisa ser usado em qualquer operação de memória, pois todas elas precisam modificar elementos da arena. Em máquinas testadas, isso gasta cerca de 40 bytes se usado.
- **Uso Máximo:** Opcional. Só precisamos definir isso se estamos rodando o programa em um nível alto de depuração e por isso queremos saber ao fim do uso da arena qual a quantidade máxima de memória que alocamos nela ao longo da execução do programa. Desta forma, se nosso programa sempre disser que usamos uma quantidade pequena demais de memória, podemos ajustar o valor para alocar menos memória. Ou se chegarmos perto demais do valor máximo de alocação, podemos mudar o valor ou depurar o programa para gastarmos menos memória. Se estivermos monitorando o valor, precisamos verificar se ele precisa ser atualizado após qualquer alocação ou criação de **breakpoint**.
- **Nome de Arquivo:** Opcional. Nome do arquivo onde a arena é criada para podermos imprimir mensagens úteis para depuração.
- **Linha:** Opcional. Número da linha em que a arena é criada. Informação usada apenas para imprimir mensagens de depuração.

Caso usemos todos estes dados, nosso cabeçalho de memória ficará com cerca de 124 bytes em máquinas típicas. Nosso cabeçalho de arena terá então a seguinte definição na linguagem C:

**Seção: Declarações de Memória (continuação):**

```
struct _arena_header{
  size_t total, used;
  struct _breakpoint *last_breakpoint;
  void *empty_position, *last_element;
#if W_DEBUG_LEVEL >= 1
  char file[32];
  unsigned long line;
#endif
#ifdef W_MULTITHREAD
  pthread_mutex_t mutex;
#endif
#if W_DEBUG_LEVEL >= 3
  size_t max_used;
#endif
};
```

Pela definição, existem algumas restrições sobre os valores presentes em cabeçalhos de arena. Vamos criar um código de depuração para testar que qualquer uma destas restrições não é violada:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 4
void _assert__arena_header(struct _arena_header *);
#endif
```

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

```
#if W_DEBUG_LEVEL >= 4
void _assert__arena_header(struct _arena_header *header){
  // O espaço máximo disponível na arena sempre deve ser maior ou
  // igual ao máximo que já armazenamos nela.
  if(header -> total < header -> max_used){
    fprintf(stderr,
            "ERROR (4): MEMORY: Arena header used more memory than allowed!\n");
    exit(1);
```

```
  }
  // Já o máximo que já armazenamos deve ser maior ou igual ao que
  // estamos armazenando no instante atual (pela definição de
  // 'máximo')
  if(header -> max_used < header -> used){
    fprintf(stderr,
            "ERROR (4): MEMORY: Arena header not registering max usage!\n");
    exit(1);
  }
  // O último breakpoint é o último elemento ou está antes do último
  // elemento. Já que breakpoints são elementos, mas há outros
  // elementos além de breakpoints.
  if((void *) header -> last_element < (void *) header -> last_breakpoint){
    fprintf(stderr,
            "ERROR (4): MEMORY: Arena header storing in wrong location!\n");
    exit(1);
  }
  // O espaço não-alocado não existe ou fica depois do último elemento
  // alocado.
  if(!(header -> empty_position == NULL ||
       (void *) header -> empty_position > (void *) header -> last_element)){
    fprintf(stderr,
            "ERROR (4): MEMORY: Arena header confused about empty position!\n");
    exit(1);
  }
  // Toda arena ocupa algum espaço, nem que sejam os bytes gastos pelo
  // cabeçalho.
  if(header -> used <= 0){
    fprintf(stderr,
            "ERROR (4): MEMORY: Arena header not occupying space!\n");
    exit(1);
  }
}
#endif
```

Quando criamos a arena e desejamos inicializar o valor de seu cabeçalho, tudo o que precisamos saber é o tamanho total que a arena tem, o nome od arquivo e número de linha. Os demais valores podem ser deduzidos. Portanto, podemos usar esta função interna para a tarefa:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
static bool _initialize_arena_header(struct _arena_header *header,
                                     size_t total
#if W_DEBUG_LEVEL >= 1
                                     , char *filename,unsigned long line
#endif
                                     ){
  header -> total = total;
  header -> used = sizeof(struct _arena_header) - sizeof(struct _breakpoint);
  header -> last_breakpoint = (struct _breakpoint *) (header + 1);
  header -> last_element = (void *) header -> last_breakpoint;
  header -> empty_position = (void *) (header -> last_breakpoint + 1);
```

```
#ifdef W_MULTITHREAD
  if(pthread_mutex_init(&(header -> mutex), NULL) != 0){
    return false;
  }
#endif
#if W_DEBUG_LEVEL >= 1
  header -> line = line;
  strncpy(header -> file, filename, 32);
#endif
#if W_DEBUG_LEVEL >= 3
  header -> max_used = header -> used;
#endif
#if W_DEBUG_LEVEL >= 4
  _assert__arena_header(header);
#endif
  return true;
}
```

É importante notar que tal função de inicialização só pode falhar se ocorrer algum erro inicializando o mutex. Por isso podemos representar o seu sucesso ou fracasso fazendo-a retornar um valor booleano.

## 2.1.2- Breakpoints

A função primária de um breakpoint é interagir com as função `Wbreakpoint` e `Wtrash`. As informações que devem estar presentes nele são:

- **Tipo:** Um número mágico que corresponde sempre à um valor que identifica o elemento como sendo um *breakpoint*, e não um fragmento alocado de memória. Se o elemento realmente for um breakpoint e não possuir um número mágico correspondente, então ocorreu um *buffer overflow* em memória alocada e podemos acusar isso. Definiremos tal número como `0x11010101`.
- **Último breakpoint:** No caso do primeiro breakpoint, isso deve apontar para ele próprio (e assim o primeiro breakpoint pode ser identificado diante dos demais). nos demais casos, ele irá apontar para o breakpoint anterior. Desta forma, em caso de `Wtrash`, poderemos restaurar o cabeçalho da arena para apontar para o breakpoint anterior, já que o atual está sendo apagado.
- **Último Elemento:** Para que a lista de elementos de uma arena possa ser percorrida, cada elemento deve ser capaz de apontar para o elemento anterior. Desta forma, se o breakpoint for removido, podemos restaurar o último elemento da arena para o elemento antes dele (assumindo que não tenha sido marcado para remoção como será visto adiante). O último elemento do primeiro breakpoint é ele próprio.
- **Arena:** Um ponteiro para a arena à qual pertence a memória.
- **Tamanho:** A quantidade de memória alocada até o breakpoint em questão. Quando o breakpoint for removido, a quantidade de memória usada pela arena passa a ser o valor presente aqui.
- **Arquivo:** Opcional para depuração. O nome do arquivo onde esta região da memória foi alocada.
- **Linha:** Opcional para depuração. O número da linha onde esta região da memória foi alocada.

Sendo assim, a nossa definição de breakpoint é:

**Seção: Declarações de Memória (continuação):**

```
struct _breakpoint{
  unsigned long type;
#if W_DEBUG_LEVEL >= 1
  char file[32];
  unsigned long line;
#endif
```

```
  void *last_element;
  struct _arena_header *arena;
  // Todo elemento dentro da memória (breakpoints e cabeçalhos de
  // memória) terão os 5 campos anteriores no mesmo local. Desta
  // forma, independente deles serem breakpoints ou regiões alocadas,
  // sempre será seguro usar um casting para qualquer um dos tipos e
  // consultar qualquer um dos 5 campos anteriores. O campo abaixo,
  // 'last_breakpoint', por outro lado, só pode ser consultado por
  // breakpoints.
  struct _breakpoint *last_breakpoint;
  size_t size;
};
```

Se todos os elementos estiverem presentes, espera-se que um *breakpoint* tenha por volta de 72 bytes. Naturalmente, isso pode variar dependendo da máquina.

As seguintes restrições sempre devem valer para tais dados:

a) $type = 0{\times}11010101$. Mas é melhor declarar uma macro para não esquecer o valor:

**Seção: Declarações de Memória (continuação):**

```
#define _BREAKPOINT_T  0x11010101
```

b) $last\_breakpoint \leq last\_element$.

Vamos criar uma função de depuração que nos ajude a checar por tais erros. O caso do tipo de um *breakpoint* não casar com o valor esperado é algo possível de acontecer principalmente devido à *buffer overflows* causados devido à erros do programador que usa a API. Por causa disso, teremos que ficar de olho em tais erros quando `W_DEBUG_LEVEL >= 1`, não penas quando `W_DEBUG_LEVEL >= 4`. Esta é a função que checa um *breakpoint* por erros:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 1
void _assert__breakpoint(struct _breakpoint *);
#endif
```

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
#if W_DEBUG_LEVEL >= 1
void _assert__breakpoint(struct _breakpoint *breakpoint){
  if(breakpoint -> type != _BREAKPOINT_T){
    fprintf(stderr,
            "ERROR (1): Probable buffer overflow. We can't guarantee a "
            "reliable error message in this case. But the "
            "data where the buffer overflow happened may be "
            "the place allocated at %s:%lu or before.\n",
            ((struct _breakpoint *)
              breakpoint -> last_element) -> file,
            ((struct _breakpoint *)
              breakpoint -> last_element) -> line);
    exit(1);
  }
#if W_DEBUG_LEVEL >= 4
  if((void *) breakpoint -> last_breakpoint >
                          (void *) breakpoint -> last_element){
    fprintf(stderr, "ERROR (4): MEMORY: Breakpoint's previous breakpoint "
                    "found after breakpoint's last element.\n");
```

```
    exit(1);
  }
#endif
}
#endif
```

Vamos agora cuidar de uma função para inicializar os valores de um breakpoint. Para isso vamos precisar saber o valor de todos os elementos, exceto o `type` e o tamanho que pode ser deduzido pela arena:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
static void _initialize_breakpoint(struct _breakpoint *self,
                                   void *last_element,
                                   struct _arena_header *arena,
                                   struct _breakpoint *last_breakpoint
#if W_DEBUG_LEVEL >= 1
                                   , char *file, unsigned long line
#endif
                                   ){
  self -> type = _BREAKPOINT_T;
  self -> last_element = last_element;
  self -> arena = arena;
  self -> last_breakpoint = last_breakpoint;
  self -> size = arena -> used - sizeof(struct _breakpoint);
#if W_DEBUG_LEVEL >= 1
  strncpy(self -> file, file, 32);
  self -> line = line;
  _assert__breakpoint(self);
#endif
}
```

Notar que assumimos que quando vamos inicializar um breakpoint, todos os dados do cabeçalho da arena já foram atualizados como tendo o breakpoint já existente. E como consultamos tais dados, o mutex da arena precisa estar bloqueado para que coisas como o tamanho da arena não mudem.

O primeiro dos breakpoints é especial e pode ser inicializado como abaixo. Para ele não precisamos nos preocupar em armazenar o nome de arquivo e número de linha em que é definido.

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
static void _initialize_first_breakpoint(struct _breakpoint *self,
                                         struct _arena_header *arena){
#if W_DEBUG_LEVEL >= 1
  _initialize_breakpoint(self, self, arena, self, "", 0);
#else
  _initialize_breakpoint(self, self, arena, self);
#endif
}
```

## 2.1.3- Memória alocada

Por fim, vamos à definição da memória alocada. Ela é formada basicamente por um cabeçalho, o espaço alocado em si e uma finalização. No caso do cabeçalho, precisamos dos seguintes elementos:

•**Tipo:** Um número que identifica o elemento como um cabeçalho de dados, não um breakpoint. No caso, usaremos o número mágico 0×10101010. Para não esquecer, é melhor definir uma macro para se referir à ele:

**Seção: Declarações de Memória (continuação):**

```
#define _DATA_T         0x10101010
```

•**Tamanho Real:** Quantos bytes tem a região alocada para dados. É igual ao tamanho pedido mais alguma quantidade adicional de bytes de preenchimento para podermos manter o alinhamento da memória.
•**Tamanho Pedido:** Quantos bytes foram pedidos na alocação, ignorando o preenchimento.
•**Último Elemento:** A posição do elemento anterior da arena. Pode ser outro cabeçalho de dado alocado ou um breakpoint. Este ponteiro nos permite acessar os dados como uma lista encadeada.
•**Arena:** Um ponteiro para a arena à qual pertence a memória. **Flags:** Permite que coloquemos informações adicionais. o último bit é usado para definir se a memória foi marcada para ser apagada ou não.
•**Arquivo:** Opcional para depuração. O nome do arquivo onde esta região da memória foi alocada.
•**Linha:** Opcional para depuração. O número da linha onde esta região da memória foi alocada.

A definição de nosso cabeçalho de dados é:

**Seção: Declarações de Memória (continuação):**

```
struct _memory_header{
  unsigned long type;
#if W_DEBUG_LEVEL >= 1
  char file[32];
  unsigned long line;
#endif
  void *last_element;
  struct _arena_header *arena;
  // Os campos acima devem ser idênticos aos 5 primeiros do 'breakpoint'
  size_t real_size, requested_size;
  unsigned long flags;
};
```

Notar que as seguintes restrições sempre devem ser verdadeiras para este cabeçalho de região alocada:

a) $type = 0\times10101010$. Ou significa que ocorreu um *buffer overflow*.

b) $real\_size \geq requested\_size$. A quantidade de bytes de preenchimento é no mínimo zero. Não iremos alocar um valor menor que o pedido.

A função que irá checar a integridade de nosso cabeçalho de memória é:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 1
void _assert__memory_header(struct _memory_header *);
#endif
```

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
#if W_DEBUG_LEVEL >= 1
void _assert__memory_header(struct _memory_header *mem){
  if(mem -> type != _DATA_T){
    fprintf(stderr,
            "ERROR (1): Probable buffer overflow. We can't guarantee a "
            "reliable error message in this case. But the "
            "data where the buffer overflow happened may be "
```

```
            "the place allocated at %s:%lu or before.\n",
            ((struct _memory_header *)
              mem -> last_element) -> file,
            ((struct _memory_header *)
              mem -> last_element) -> line);
    exit(1);
  }
#if W_DEBUG_LEVEL >= 4
  if(mem -> real_size < mem -> requested_size){
    fprintf(stderr,
            "ERROR (4): MEMORY: Allocated less memory than requested in "
            "data allocated in %s:%lu.\n", mem -> file, mem -> line);
    exit(1);
  }
#endif
}
#endif
```

Não criaremos uma função de inicialização para este cabeçalho. Ele será inicializado dentro da função que aloca mais espaço na memória. Ao contrário de outros cabeçalhos, não há nenhuma facilidade em criar um inicializador para este, pois todos os dados a serem inicializados precisam ser passados explicitamente. Nada pode ser meramente deduzido, exceto o `real_size`. Mas de qualquer forma o `real_size` precisa ser calculado antes do preenchimento do cabeçalho, para atualizar o cabeçalho da própria arena.

## 2.2 - Criando e destruindo arenas

Criar uma nova arena envolve basicamente alocar memória usando `mmap` e tomando o cuidado para alocarmos sempre um número múltiplo do tamanho de uma página (isso garante alinhamento de memória e também nos dá um tamanho ótimo para paginarmos). Em seguida preenchemos o cabeçalho da arena e colocamos o primeiro breakpoint nela.

A função que cria novas arenas deve receber como argumento o tamanho mínimo que ela deve ter em bytes. Já destruir uma arena requer um ponteiro para ela:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 1
  // Se estamos em modo de depuração, a função precisa estar ciente do
  // nome do arquivo e linha em que é invocada:
void *_Wcreate_arena(size_t size, char *filename, unsigned long line);
#define Wcreate_arena(a) _Wcreate_arena(a, __FILE__, __LINE__)
#else
void *_Wcreate_arena(size_t size);
#define Wcreate_arena(a) _Wcreate_arena(a)
#endif
int Wdestroy_arena(void *);
```

### 2.2.1- Criando uma arena

O processo de criar a arena funciona alocando todo o espaço de que precisamos e em seguida preenchendo o cabeçalho inicial e breakpoint:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
  // Os argumentos que a função recebe são diferentes no modo de
```

```
  // depuração e no modo final:
#if W_DEBUG_LEVEL >= 1
void *_Wcreate_arena(size_t size, char *filename, unsigned long line){
#else
void *_Wcreate_arena(size_t size){
#endif
  void *arena;
  size_t real_size = 0;
  struct _breakpoint *breakpoint;
  // Aloca arena calculando seu tamanho verdadeiro à partir do tamanho pedido:
  long page_size = sysconf(_SC_PAGESIZE);
  real_size = ((int) ceil((double) size / (double) page_size)) * page_size;
  arena = mmap(0, real_size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
               -1, 0);
  if(arena == MAP_FAILED)
    arena = NULL; // Se algo falha, retornamos NULL
  if(arena != NULL){
    if(!_initialize_arena_header((struct _arena_header *) arena, real_size
#if W_DEBUG_LEVEL >= 1
  ,filename, line // Dois argumentos a mais em modo de depuração
#endif
  )){
      // Se não conseguimos inicializar o cabeçalho da arena,
      // desalocamos ela com munmap:
      munmap(arena, ((struct _arena_header *) arena) -> total);
      // O munmap pode falhar, mas não podemos fazer nada à este
      // respeito.
      return NULL;
    }
    // Preenchendo o primeiro breakpoint
    breakpoint = ((struct _arena_header *) arena) -> last_breakpoint;
    _initialize_first_breakpoint(breakpoint, (struct _arena_header *) arena);
#if W_DEBUG_LEVEL >= 4
    _assert__arena_header(arena);
#endif
  }
  return arena;
}
```

Então usar esta função nos dá como retorno `NULL` ou um ponteiro para uma nova arena cujo tamanho total é no mínimo o pedido como argumento, mas talvez seja maior por motivos de alinhamento e paginação. Partes desta região contínua serão gastos com cabeçalhos da arena, das regiões alocadas e *breakpoints*. Então pode ser que obtenhamos como retorno uma arena onde caibam menos coisas do que caberia no tamanho especificado como argumento.

O tamanho fnal que a arena terá para colocar todas as coisas será o menor múltiplo de uma página do sistema que pode conter o tamanho pedido.

Usamos `sysconf` para saber o tamanho da página e `mmap` para obter a memória. Outra opção seria o `brk`, mas usar tal chamada de sistema criaria conflito caso o usuário tentasse usar o `malloc` da biblioteca padrão ou usasse uma função de biblioteca que usa internamente o `malloc`. Como até um simples `sprintf` usa `malloc`, não é prático usar o `brk`, pois isso criaria muitos conflitos com outras bibliotecas.

## 2.2.2- Checando vazamento de memória em uma arena

Uma das grandes vantagens de estarmos cuidando do gerenciamento de memória é podermos checar a existência de vazamentos de memória no fim do programa. Recapitulando, uma arena de memória ao ser alocada conterá um cabeçalho de arena, um *breakpoint* inicial e por fim, tudo aquilo que foi alocada nela (que podem ser dados de memória ou outros *breakpoints*). Sendo assim, se depois de alocar tudo com o nosso `Walloc` (que ainda iremos definir) nós desalocarmos com o nosso `Wfree` ou `Wtrash` (que também iremos definir), no fim a arena ficará vazia sem nada após o primeiro *breakpoint*. Exatamente como quando a arena é recém-criada.

Então podemos inserir código que checa para nós se isso realmente é verdade e que pode ser invocado sempre antes de destruirmos uma arena. Se encontrarmos coisas na memória, isso significa que o usuário alocou memória e não desalocou. Caberá ao nosso código então imprimir uma mensagem de depuração informando do vazamento de memória e dizendo em qual arquivo e número de linha ocorreu a tal alocação.

A função que fará isso para nós será:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 1
void _assert_no_memory_leak(void *);
#endif
```

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
#if W_DEBUG_LEVEL >= 1
void _assert_no_memory_leak(void *arena){
  struct _arena_header *header = (struct _arena_header *) arena;
  // Primeiro vamos para o último elemento da arena
  struct _memory_header *p = (struct _memory_header *) header -> last_element;
  // E vamos percorrendo os elementos de trás pra frente imprimindo
  // mensagem de depuração até chegarmos no breakpoint inicial (que
  // aponta para ele mesmo como último breakpoint):
  while(p -> type != _BREAKPOINT_T ||
        ((struct _breakpoint *) p) -> last_breakpoint !=
        (struct _breakpoint *) p){
    if(p -> type == _DATA_T && p -> flags % 2){
      fprintf(stderr, "WARNING (1): Memory leak in data allocated in %s:%lu\n",
              p -> file, p -> line);
    }
    p = (struct _memory_header *) p -> last_element;
  }
}
#endif
```

Esta função será usada automaticamente desde que estejamos compilando uma versão de desenvolvimento do jogo. Entretanto, não há nenhum modo de realmente garantirmos que toda arena criada será destruída. Se ela não for, independente dela conter ou não coisas ainda alocadas, isso será um vazamento não-detectado.

## 2.2.3- Destruindo uma arena

Destruir uma arena é uma simples questão de finalizar o seu mutex caso estejamos criando um programa com muitas threads e usar um `munmap` . Também é quando invocamos a checagem por vazamento de memória e dependendo do nível da depuração, podemos imprimir também a quantidade máxima de memória usada:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
int Wdestroy_arena(void *arena){
#if W_DEBUG_LEVEL >= 1
  _assert_no_memory_leak(arena);
#endif
#if W_DEBUG_LEVEL >= 4
    _assert__arena_header(arena);
#endif
#if W_DEBUG_LEVEL >= 3
  fprintf(stderr,
          "WARNING (3): Max memory used in arena %s:%lu: %lu/%lu\n",
          ((struct _arena_header *) arena) -> file,
          ((struct _arena_header *) arena) -> line,
          (unsigned long) ((struct _arena_header *) arena) -> max_used,
          (unsigned long) ((struct _arena_header *) arena) -> total);
#endif
#ifdef W_MULTITHREAD
  {
    struct _arena_header *header = (struct _arena_header *) arena;
    if(pthread_mutex_destroy(&(header -> mutex)) != 0)
      return 0;
  }
#endif
  //Desaloca 'arena'
  if(munmap(arena, ((struct _arena_header *) arena) -> total) == -1)
    arena = NULL;
  if(arena == NULL) return 0;
  else return 1;
}
```

## 2.3 - Alocação e desalocação de memória

Agora chegamos à parte mais usada de um gerenciador de memórias: alocação e desalocação. A função de alocação deve receber um ponteiro para a arena onde iremos alocar e qual o tamanho a ser alocado. A função de desalocação só precisa receber o ponteiro da região a ser desalocada, pois informações sobre a arena serão encontradas em seu cabeçalho imediatamente antes da região de uso da memória. Dependendo do nível de depuração, ambas as funções precisam também saber de que arquivo e número de linha estão sendo invocadas e isso justifica o forte uso de macros abaixo:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 1
  void *_alloc(void *arena, size_t size, char *filename, unsigned long line);
#define Walloc_arena(a, b) _alloc(a, b, __FILE__, __LINE__)
#else
  void *_alloc(void *arena, size_t size);
#define Walloc_arena(a, b) _alloc(a, b)
#endif
#if W_DEBUG_LEVEL >= 2
  void _free(void *mem, char *filename, unsigned long line);
#define Wfree(a) _free(a, __FILE__, __LINE__)
#else
```

```
  void _free(void *mem);
#define Wfree(a) _free(a)
#endif
```

Ao alocar memória, precisamos ter a preocupação de manter um alinhamento de bytes para não prejudicar o desempenho. Por causa disso, às vezes precisamos alocar mais que o pedido. Por exemplo, se o usuário pede para alocar somente 1 byte, podemos precisar alocar 3 bytes adicionais além dele só para manter o alinhamento de 4 bytes de dados. O tamanho que usamos como referência para o alinhamento é o tamanho de um `long`. Sempre alocamos valores múltiplos de um `long` que sejam suficientes para conter a quantidade de bytes pedida.

Se estamos trabalhando com múltiplas threads, precisamos também garantir que o mutex da arena em que estamos seja bloqueado, pois temos que mudar valores da arena para indicar que estamos ocupando mais espaço nela.

Por fim, se tudo deu certo basta preenchermos o cabeçalho da região de dados da arena que estamos criando. E ao retornar, retornaremos um ponteiro para o início da região que o usuário pode usar para armazenamento (e não da região que contém o cabeçalho). Se alguma coisa falhar (pode não haver mais espaço suficiente na arena) precisamos retornar `NULL` e dependendo do nível de depuração, imprimimos uma mensagem de aviso.

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
#if W_DEBUG_LEVEL >= 1
void *_alloc(void *arena, size_t size, char *filename, unsigned long line){
#else
void *_alloc(void *arena, size_t size){
#endif
  struct _arena_header *header = arena;
  struct _memory_header *mem_header;
  void *mem = NULL, *old_last_element;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&(header -> mutex));
#endif
#if W_DEBUG_LEVEL >= 4
    _assert__arena_header(arena);
#endif
  mem_header = header -> empty_position;
  old_last_element = header -> last_element;
  // Calcular o verdadeiro tamanho múltiplo de 'long' a se alocar:
  size_t real_size = (size_t) (ceil((float) size / (float) sizeof(long)) *
                               sizeof(long));
  if(header -> used + real_size + sizeof(struct _memory_header) >
     header -> total){
    // Chegamos aqui neste 'if' se não há memória suficiente
#if W_DEBUG_LEVEL >= 1
    fprintf(stderr, "WARNING (1): No memory enough to allocate in %s:%lu.\n",
      filename, line);
#endif
#ifdef W_MULTITHREAD
    pthread_mutex_unlock(&(header -> mutex));
#endif
    return NULL;
  }
  // Atualizando o cabeçalho da arena
```

```
  header -> used += real_size + sizeof(struct _memory_header);
  mem = (void *) ((char *) header -> empty_position +
                  sizeof(struct _memory_header));
  header -> last_element = header -> empty_position;
  header -> empty_position = (void *) ((char *) mem + real_size);
#if W_DEBUG_LEVEL >= 3
  // Se estamos tomando nota do máximo de memória que usamos:
  if(header -> used > header -> max_used)
    header -> max_used = header -> used;
#endif
  // Preenchendo o cabeçalho do dado a ser alocado. Este cabeçalho
  // fica imediatamente antes do local cujo ponteiro retornamos para o
  // usuário usar:
  mem_header -> type = _DATA_T;
  mem_header -> last_element = old_last_element;
  mem_header -> real_size = real_size;
  mem_header -> requested_size = size;
  mem_header -> flags = 0x1;
  mem_header -> arena = arena;
#if W_DEBUG_LEVEL >= 1
  strncpy(mem_header -> file, filename, 32);
  mem_header -> line = line;
  _assert__memory_header(mem_header);
#endif
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(header -> mutex));
#endif
  return mem;
}
```

Para desalocar a memória, existem duas possibilidades. Podemos estar desalocando a última memória alocada ou não. No primeiro caso, tudo é uma questão de atualizar o cabeçalho da arena modificando o valor do último elemento armazenado e também um ponteiro pra o próximo espaço vazio. No segundo caso, tudo o que fazemos é marcar o elemento para ser desalocado no futuro sem desalocá-lo de verdade no momento.

Não podemos desalocar sempre porque nosso espaço de memória é uma pilha. Os elementos só podem ser desalocados de verdade na ordem inversa em que são alocados. Quando isso não ocorre, a memória começa a se fragmentar ficando com buracos internos que não podem ser usados até que os elementos que vem depois não sejam também desalocados.

Isso pode parecer ruim, mas se a memória do projeto for bem-gerenciada pelo programador, não chegará a ser um problema e ficamos com um gerenciamento mais rápido. Se o programador preferir, ele tambéem pode usar o `malloc` da biblioteca padrão para não ter que se preocupar com a ordem de desalocações. Uma discussão sobre as consequências de cada caso pode ser encontrada ao fim deste capítulo.

Se nós realmente desalocamos a memória, pode ser que antes dela encontremos regiões que já foram marcadas para ser desalocadas, mas ainda não foram. É neste momento em que realmente as desalocamos eliminando a fragmentação naquela parte.

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
#if W_DEBUG_LEVEL >= 2
void _free(void *mem, char *filename, unsigned long line){
#else
void _free(void *mem){
```

```
#endif
  struct _memory_header *mem_header = ((struct _memory_header *) mem) - 1;
  struct _arena_header *arena = mem_header -> arena;
  void *last_freed_element;
  size_t memory_freed = 0;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&(arena -> mutex));
#endif
#if W_DEBUG_LEVEL >= 4
    _assert__arena_header(arena);
#endif
#if W_DEBUG_LEVEL >= 1
    _assert__memory_header(mem_header);
#endif
  // Primeiro checamos se não estamos desalocando a ultima memória. Se
  // não é a ultima memória, não precisamos manter o mutex ativo e
  // apenas marcamos o dado presente para ser desalocado no futuro.
  if((struct _memory_header *) arena -> last_element != mem_header){
#ifdef W_MULTITHREAD
    pthread_mutex_unlock(&(arena -> mutex));
#endif
    mem_header -> flags = 0x0;
#if W_DEBUG_LEVEL >= 2
  // Pode ser que tenhamos que imprimir um aviso de depuração acusando
  // desalocação na ordem errada:
    fprintf(stderr,
            "WARNING (2): %s:%lu: Memory allocated in %s:%lu should be"
            " freed first to prevent fragmentation.\n", filename, line,
            ((struct _memory_header *) (arena -> last_element)) -> file,
            ((struct _memory_header *) (arena -> last_element)) -> line);
#endif
    return;
  }
  // Se estamos aqui, esta é uma desalocação verdadeira. Calculamos
  // quanto espaço iremos liberar:
  memory_freed = mem_header -> real_size + sizeof(struct _memory_header);
  last_freed_element = mem_header;
  mem_header = mem_header -> last_element;
  // E também levamos em conta que podemos desalocar outras coisas que
  // tinham sido marcadas para ser desalocadas:
  while(mem_header -> type != _BREAKPOINT_T && mem_header -> flags == 0x0){
    memory_freed += mem_header -> real_size + sizeof(struct _memory_header);
    last_freed_element = mem_header;
    mem_header = mem_header -> last_element;
  }
  // Terminando de obter o tamanho total a ser desalocado e obter
  // novos valores para ponteiros, atualizamos o cabeçalho da arena:
  arena -> last_element = mem_header;
  arena -> empty_position = last_freed_element;
```

```
  arena -> used -= memory_freed;
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(arena -> mutex));
#endif
}
```

## 2.4 - Usando a heap descartável

Graças ao conceito de *breakpoints*, pode-se desalocar ao mesmo tempo todos os elementos alocados desde o último *breakpoint* por meio do `Wtrash`. A criação de um *breakpoit* e descarte de memória até ele se dá por meio das funções declaradas abaixo:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 1
int _new_breakpoint(void *arena, char *filename, unsigned long line);
#define Wbreakpoint_arena(a) _new_breakpoint(a, __FILE__, __LINE__)
#else
int _new_breakpoint(void *arena);
#define Wbreakpoint_arena(a) _new_breakpoint(a)
#endif
void Wtrash_arena(void *arena);
```

As funções precisam receber como argumento apenas um ponteiro para a arena na qual realizar a operação. Além disso, dependendo do nível de depuração, elas recebem também o nome de arquivo e número de linha como nos casos anteriores para que isso ajude na depuração:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
#if W_DEBUG_LEVEL >= 1
int _new_breakpoint(void *arena, char *filename, unsigned long line){
#else
int _new_breakpoint(void *arena){
#endif
  struct _arena_header *header = (struct _arena_header *) arena;
  struct _breakpoint *breakpoint, *old_breakpoint;
  void *old_last_element;
  size_t old_size;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&(header -> mutex));
#endif
#if W_DEBUG_LEVEL >= 4
    _assert__arena_header(arena);
#endif
  if(header -> used + sizeof(struct _breakpoint) > header -> total){
    // Se estamos aqui, não temos espaço para um breakpoint
#if W_DEBUG_LEVEL >= 1
    fprintf(stderr, "WARNING (1): No memory enough to allocate in %s:%lu.\n",
      filename, line);
#endif
#ifdef W_MULTITHREAD
    pthread_mutex_unlock(&(header -> mutex));
#endif
```

```
    return 0;
  }
  // Atualizando o cabeçalho da arena e salvando valores relevantes
  old_breakpoint = header -> last_breakpoint;
  old_last_element = header -> last_element;
  old_size = header -> used;
  header -> used += sizeof(struct _breakpoint);
  breakpoint = (struct _breakpoint *) header -> empty_position;
  header -> last_breakpoint = breakpoint;
  header -> empty_position = ((struct _breakpoint *) header -> empty_position) +
    1;
  header -> last_element = header -> last_breakpoint;
#if W_DEBUG_LEVEL >= 3
  if(header -> used > header -> max_used){ // Batemos récorde de uso?
    header -> max_used = header -> used;
  }
#endif
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(header -> mutex));
#endif
  breakpoint -> type = _BREAKPOINT_T; // Preenchendo cabeçalho do breakpoint
  breakpoint -> last_element = old_last_element;
  breakpoint -> arena = arena;
  breakpoint -> last_breakpoint = (void *) old_breakpoint;
  breakpoint -> size = old_size;
#if W_DEBUG_LEVEL >= 1
  strncpy(breakpoint -> file, filename, 32);
  breakpoint -> line = line;
#endif
#if W_DEBUG_LEVEL >= 4
  _assert__breakpoint(breakpoint);
#endif
  return 1;
}
```

E a função para descartar toda a memória presente na heap até o último breakpoint:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
void Wtrash_arena(void *arena){
  struct _arena_header *header = (struct _arena_header *) arena;
  struct _breakpoint *previous_breakpoint =
    ((struct _breakpoint *) header -> last_breakpoint) -> last_breakpoint;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&(header -> mutex));
#endif
#if W_DEBUG_LEVEL >= 4
    _assert__arena_header(arena);
    _assert__breakpoint(header -> last_breakpoint);
#endif
  if(header -> last_breakpoint == previous_breakpoint){
```

```
    // Chegamos aqui se existe apenas 1 breakpoint
    header -> last_element = previous_breakpoint;
    header -> empty_position = (void *) (previous_breakpoint + 1);
    header -> used = previous_breakpoint -> size + sizeof(struct _breakpoint);
  }
  else{
    // Chegamos aqui se há 2 ou mais breakpoints
    struct _breakpoint *last = (struct _breakpoint *) header -> last_breakpoint;
    header -> used = last -> size;
    header -> empty_position = last;
    header -> last_element = last -> last_element;
    header -> last_breakpoint = previous_breakpoint;
  }
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&(header -> mutex));
#endif
}
```

A função acima é totalmente inócua se não existem dados a serem desalocados até o último *breakpoint*. Neste caso ela simplesmente apaga o *breakpoint* se ele não for o único, e não faz nada se existe apenas o *breakpoint* inicial.

## 2.5 - Usando as arenas de memória padrão

Ter que se preocupar com arenas geralmente é desnecessário. O usuário pode querer simplesmente usar uma função `Walloc` sem ter que se preocupar com qual arena usar. Weaver simplesmente assumirá a existência de uma arena padrão e associada à ela as novas funções `Wfree`, `Wbreakpoint` e `Wtrash`.

Primeiro precisaremos declarar duas variáveis globais. Uma delas será uma arena padrão do usuário, a outra deverá ser uma arena usada pelas funções internas da própria API. Ambas as variáveis devem ficar restritas ao módulo de memória, então serão declaradas como estáticas:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
static void *_user_arena, *_internal_arena;
```

Noe que elas serão variáveis estáticas. Isso garantirá que somente as funções que definiremos aqui poderão manipulá-las. Será impossível mudá-las ou usá-las sem que seja usando as funções relacionadas ao gerenciador de memória. Vamos precisar inicializar e finalizar estas arenas com as seguinte funções:

**Seção: Declarações de Memória (continuação):**

```
void _initialize_memory();
void _finalize_memory();
```

Que são definidas como:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
void _initialize_memory(void){
  _user_arena = Wcreate_arena(W_MAX_MEMORY);
  _internal_arena = Wcreate_arena(4000); // Cerca de 1 página
}
void _finalize_memory(){
  Wdestroy_arena(_user_arena);
  Wtrash_arena(_internal_arena);
  Wdestroy_arena(_internal_arena);
```

```
}
```

Passamos adiante o número de linha e nome do arquivo para a função de criar as arenas. Isso ocorre porque um usuário nunca invocará diretamente estas funções. Quem vai chamar tal função é a função de inicialização da API. Se uma mensagem de erro for escrita, ela deve conter o nome de arquivo e número de linha onde está a própria função de inicialização da API. Não onde tais funções estão definidas.

A invocação destas funções se dá na inicialização da API, a qual é mencionada na Introdução. Da mesma forma, na finalização da API, chamamos a função de finalização:

**Seção: API Weaver: Inicialização (continuação):**

```
_initialize_memory();
```

**Seção: API Weaver: Finalização (continuação):**

```
// Em "desalocações" desalocamos memória alocada com |Walloc|:
                <Seção a ser Inserida: API Weaver: Desalocações>
// Só então podemos finalizar o gerenciador de memória:
_finalize_memory();
```

Agora para podermos alocar e desalocar memória da arena padrão e da arena interna, criaremos a seguinte funções:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 1
void *_Walloc(size_t size, char *filename, unsigned int line);
#define Walloc(n) _Walloc(n, __FILE__, __LINE__)
void *_Winternal_alloc(size_t size, char *filename, unsigned int line);
#define _iWalloc(n) _Winternal_alloc(n, __FILE__, __LINE__)
#else
void *_Walloc(size_t size);
#define Walloc(n) _Walloc(n)
void *_Winternal_alloc(size_t size);
#define _iWalloc(n) _Winternal_alloc(n)
#endif
```

Destas o usuário irá usar mesmo a `Walloc`. A `_iWalloc` será usada apenas internamente para usarmos a arena de alocações internas da API. E precisamos que elas sejam definidas como funções, não como macros para poderem manipular as arenas, que são variáveis estáticas à este capítulo.

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
#if W_DEBUG_LEVEL >= 1
void *_Walloc(size_t size, char *filename, unsigned int line){
  return _alloc(_user_arena, size, filename, line);
}
void *_Winternal_alloc(size_t size, char *filename, unsigned int line){
  return _alloc(_internal_arena, size, filename, line);
}
#else
void *_Walloc(size_t size){
  return _alloc(_user_arena, size);
}
void *_Winternal_alloc(size_t size){
  return _alloc(_internal_arena, size);
```

```
}
#endif
```

O `Wfree` já foi definido e irá funcionar sem problemas, independente da arena à qual pertence o trecho de memória alocado. Sendo assim, resta declarar apenas o `Wbreakpoint` e `Wtrash`:

**Seção: Declarações de Memória (continuação):**

```
#if W_DEBUG_LEVEL >= 1
int _Wbreakpoint(char *filename, unsigned long line);
#define Wbreakpoint() _Wbreakpoint(__FILE__, __LINE__)
#else
int _Wbreakpoint();
#define Wbreakpoint() _Wbreakpoint()
#endif
void _Wtrash();
#define Wtrash() _Wtrash()
```

A definição das funções segue abaixo:

**Arquivo:** `project/src/weaver/memory.c` **(continuação):**

```
#if W_DEBUG_LEVEL >= 1
int _Wbreakpoint(char *filename, unsigned long line){
  return _new_breakpoint(_user_arena, filename, line);
}
#else
int _Wbreakpoint(){
  return _new_breakpoint(_user_arena);
}
#endif
void _Wtrash(){
  Wtrash_arena(_user_arena);
}
```

## 2.6 - Medindo o desempenho

Existem duas macros que são úteis de serem definidas que podem ser usadas para avaliar o desempenho do gerenciador de memória definido aqui. Elas são:

**Seção: Cabeçalhos Weaver (continuação):**

```
#include <stdio.h>
#include <sys/time.h>

#define W_TIMER_BEGIN() { struct timeval _begin, _end; \
gettimeofday(&_begin, NULL);
#define W_TIMER_END() gettimeofday(&_end, NULL); \
printf("%ld us\n", (1000000 * (_end.tv_sec - _begin.tv_sec) + \
_end.tv_usec - _begin.tv_usec)); \
}
```

Como a primeira macro inicia um bloco e a segunda termina, ambas devem ser sempre usadas dentro de um mesmo bloco de código, ou um erro ocorrerá. O que elas fazem nada mais é do que usar `gettimeofday` e usar a estrutura retornada para calcular quantos microssegundos se passaram entre uma invocação e outra. Em seguida, escreve-se na saída padrão quantos microssegundos se passaram.

Como exemplo de uso das macros, podemos usar a seguinte função `main` para obtermos uma medida de performance das funções `Walloc` e `Wfree`:

**Arquivo:** `/tmp/dummy.c`:

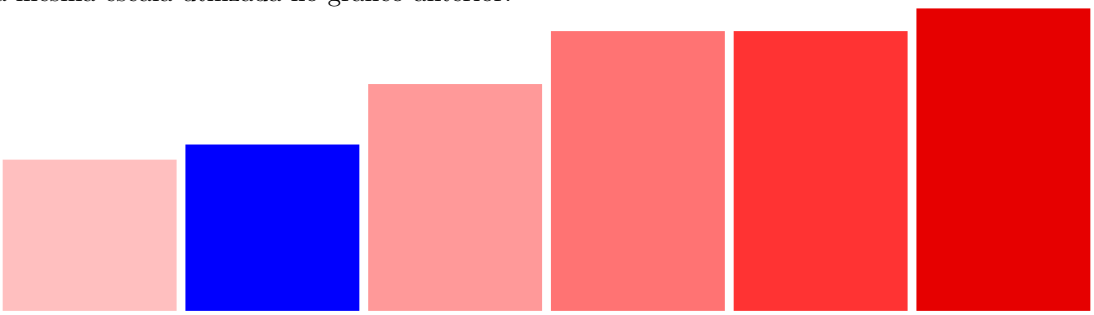
```
// Só um exemplo, não faz parte de Weaver
#include "game.h"
#define T 1000000

int main(int argc, char **argv){
  long i;
  void *m[T];
  Winit();
  W_TIMER_BEGIN();
  for(i = 0; i < T; i ++){
    m[i] = Walloc(1);
  }
  for(i = T-1; i >=0; i --){
    Wfree(m[i]);
  }
  Wtrash();
  W_TIMER_END();
  Wexit();
  return 0;
}
```

Rodando este código em um Pentium B980 2.40GHz Dual Core, este é o gráfico que representa o teste de desempenho. As barras vermellhas representam o uso de `Walloc` / `free` em diferentes níveis de depuração (0 é o mais claro e 4 é o mais escuro). Para comparar, em azul podemos ver o tempo gasto pelo `malloc` / `free` da biblioteca C GNU versão 2.20.

Isso nos mostra que se compilarmos nosso código sem nenhum recurso de depuração (como é feito ao compilarmos a versão final), obtemos um desempenho duas vezes mais rápido que do `malloc`.

E se alocássemos quantidades maiores que 1 byte? Se no programa acima estivéssemos alocando um milhão de fragmentos de 100 bytes? O próximo gráfico mostra este caso usando exatamente a mesma escala utilizada no gráfico anterior:



A diferença não é explicada somente pela diminuição da localidade espacial dos dados acessados. Se diminuirmos o número de alocações para somente dez mil, mantendo um total alocado de 1 MB, ainda assim o `malloc` ficaria na mesma posição se comparado ao `Walloc`. O que significa que alocando quantias maiores, o `malloc` é apenas ligeiramente pior que o `Walloc` sem recursos de depuração. Mas a diferença é apenas marginal.

E se ao invés de desalocarmos memória com `Wfree`, usássemos o `Wtrash` para desalocar tudo de uma só vez? O gráfico abaixo mostra este caso para quando alocamos 1 milhão de espaços de 1 byte usando a mesma escala:

O alto desempenho de nosso gerenciador de memória neste caso é compreensível. Podemos substituir um milhão de chamadas para uma função por uma só. Enquanto isso o `malloc` não tem esta opção e precisa chamar uma função de desalocação para cada função de alocação usada. E se usarmos isto para alocar 1 milhão de fragmentos de 100 bytes, o teste em que o `malloc` teve um desempenho semelhante ao nosso? A resposta é o gráfico:

Via de regra podemos dizer que o desempenho do `malloc` é semelhante ao do `Walloc` quando `W_DEBUG_MODE` é igual à 1. Mas quando o `W_DEBUG_MODE` é zero, obtemos sempre um desempenho melhor (embora em alguns casos a diferença possa ser marginal). Para analizar um caso em que o `Walloc` realmente se sobressai, vamos observar o comportamento quando compilamos o nosso teste de alocar 1 byte um milhão de vezes para Javascript via Emscripten (versão 1.34). O gráfico à seguir usará uma escala diferente.

O gráfico anterior se fosse desenhado usando a mesma escala dos outros, teria que ter barras com tamanho dez vezes maior. Enquanto o `Walloc` tem uma velocidade 1,8 vezes menor compilado com Emscripten, o `malloc` tem uma velocidade 20 vezes menor. Se tentarmos fazer no Emscripten o teste em que alocamos 100 bytes ao invés de 1 byte, o resultado reduzido em dez vezes fica praticamente igual ao gráfico acima.

Este é um caso no qual o `Walloc` se sobressai. Mas há também um caso em que o `Walloc` é muito pior: quando usamos várias threads. Considere o código abaixo:

**Arquivo:** `/tmp/dummy.c`:

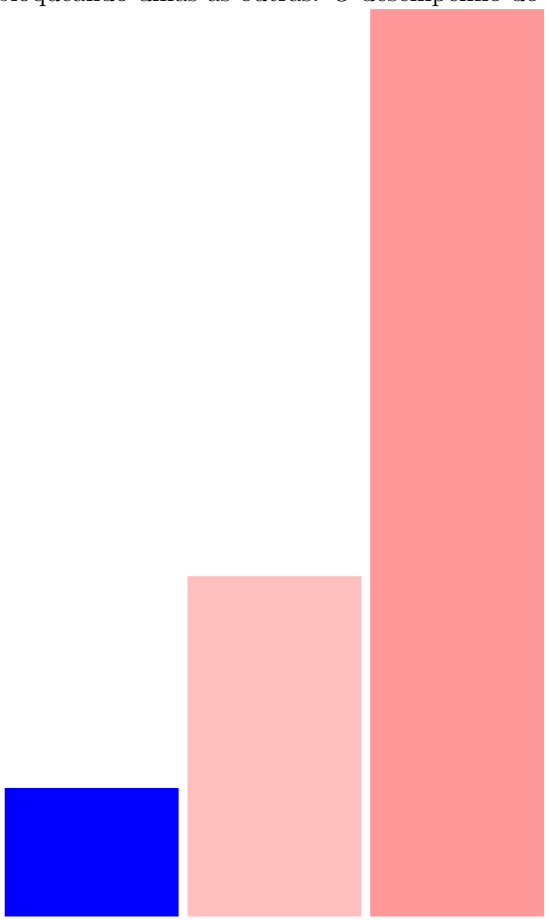
```
// Só um exemplo, não faz parte de Weaver
#define NUM_THREADS 10
#define T (1000000 / NUM_THREADS)

void *test(void *a){
  long *m[T];
  long i;
  for(i = 0; i < T; i ++){
    m[i] = (long *) Walloc(1);
    *m[i] = (long) m[i];
  }
  for(i = T-1; i >=0; i --){
    Wfree(m[i]);
  }
```

```
}

int main(void){
  pthread_t threads[NUM_THREADS];
  int i;
  Winit();
  for(i = 0; i < NUM_THREADS; i ++)
    pthread_create(&threads[i], NULL, test, (void *) NULL);
  W_TIMER_BEGIN();
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join(threads[i], NULL);
  W_TIMER_END();
  Wexit();
  pthread_exit(NULL);
  return 0;
}
```

Neste caso, assumindo que estejamos compilando com a macro `W_MULTITHREAD` no arquivo `conf/conf.h`, as threads estarão sempre competindo pela arena e passarão boa parte do tempo bloqueando umas às outras. O desempenho do `Walloc` e `malloc` neste caso será:



Para `W_DEBUG_MODE` valendo dois e três, a barra torna-se vinte vezes maior do que a vista ao lado.

Para o nível quatro de depuração, a barra torna-se quarenta vezes maior que a vista ao lado.

Neste caso, o correto seria criar uma arena para cada thread com `Wcreate_arena` , sempre fazer cada thread alocar dentro de sua arena com `Walloc_arena` , criar *breakpoints* com `Wbreakpoint_arena` , desalocar com `Wfree_arena` e descartar a heap até o último *breakpoint* com `Wtrash_arena` . Por fim, cada thread deveria finalizar sua arena com `Wdestroy_arena` . Assim poderia-se usar o desempenho maior do `Walloc` aproveitando-o melhor entre todas as threads. Pode nem ser necessário definir `W_MULTITHREAD` se as threads forem bem especializadas e não

disputarem recursos.

A nova função de teste que usamos passa a ser:

**Arquivo:** `/tmp/dummy.c`:

```
// Só um exemplo, não faz parte de Weaver
void *test(void *a){
  long *m[T];
  long i;
  void *arena = Wcreate_arena(10000000);
  for(i = 0; i < T; i ++){
    m[i] = (long *) Walloc_arena(arena, 1);
    *m[i] = (long) m[i];
  }
  for(i = T-1; i >= 0; i --){
    Wfree(m[i]);
  }
  Wtrash_arena(arena);
  Wdestroy_arena(arena);
  return NULL;
}
```

Neste caso, o gráfico de desempenho em um computador com dois processadores é:

Infelizmente não poderemos fazer os testes de threads na compilação via Emscripten. Até o momento, este é um recurso disponível somente no Firefox Nightly.

Os testes nos mostram que embora o `malloc` da biblioteca C GNU seja bem otimizado, é possível obter melhoras significativas em código compilado via Emscripten e código feito para várias threads tendo um gerenciador de memórias mais simples e personalizado. Isto e a habilidade de detectar vazamentos de memória em modo de depuração é o que justifica a criação de um gerenciador próprio para Weaver. Como a prioridade em nosso gerenciador é a velocidade, o seu uso correto para evitar fragmentação excessiva depende de conhecimento e cuidados maiores por parte do programador. Por isso espera-se que programadores menos experientes continuem usando o `malloc` enquanto o `Walloc` será usado internamente pela nossa engine e estará à disposição daqueles que querem pagar o preço por ter um desempenho maior, especialmente em certos casos específicos.

## 2.7 - O gerenciador de memória e a estrutura `W`

No Capíulo 1 dissemos que iríamos evitar definir variáveis e funções globalmente, exceto por funções e variáveis internas que tem seus nomes começando com " `_` ". Mas neste capítulo nós pareemos ignorar isso e definimos ao todo 9 funções (algumas delas como macros) fora de `W` .

O motivo de estarmos fazendo isso aqui é a excepcionalidade de algumas funções do gerenciador de memória. Para que sejamos capazes de usar recursos de depuração, como informar o número da linha em que surgiu um vazamento de memória, somos obrigados a definir algumas funções como macros para que elas usem as informações de `__LINE__` e `__FILE__` .

Mas para que tais funções também estejam disponíveis dentro de `W` para poderem ser usadas em *plugins*, vamos definir e declarar versões especiais delas feitas para isso. assim, além de `Walloc` , teremos também um `W.alloc` , além de `Wfree` teremos um `W.free` e assim por diante.

As funções dentro da estrutura serão quase idênticas. Mas elas não poderão fornecer recursos de depuração e por isso não são tão recomendadas, exceto para serem usadas dentro de *plugins*.

## 2.8 - Sumário das Variáveis e Funções de Memória

# Capítulo 3: Criando uma Janela

Para que tenhamos um jogo, precisamos de gráficos. E também precisamos de um local onde desenharmos os gráficos. Em um jogo compilado para Desktop, tipicamente criaremos uma janela na qual invocaremos funções OpenGL. Em um jogo compilado para a Web, tudo será mais fácil, pois não precisaremos de uma janela especial. Por padrão já teremos um *canvas* para manipular com WebGL. Portanto, o código para estes dois cenários irá diferir bastante neste capítulo. De qualquer forma, ambos usarão OpenGL:

**Seção: Cabeçalhos Weaver (continuação):**

```
#include <GL/glew.h>
```

Para criar uma janela, usaremos o Xlib ao invés de bibliotecas de mais alto nível. Primeiro porque muitas bibliotecas de alto nível como SDL parecem ter problemas em ambientes gráficos mais excêntricos como o *ratpoison* e *xmonad*, as quais eu uso. Particularmente recursos como tela cheia em alguns ambientes não funcionam. Ao mesmo tempo, o Xlib é uma biblioteca bastante universal. Se um sistema não tem o X, é porque ele não tem interface gráfica e não iria rodar um jogo mesmo.

O nosso arquivo `conf/conf.h` precisará de duas macros novas para estabelecermos o tamanho de nossa janela (ou do "canvas" para a Web):

- `W_DEFAULT_COLOR` : A cor padrão da janela, a ser exibida na ausência de qualquer outra coisa para desenhar. Representada como três números em ponto flutuante separados por vírgulas.
- `W_HEIGHT` : A altura da janela ou do "canvas". Se for definido como zero, será o maior tamanho possível.
- `W_WIDTH` : A largura da janela ou do "canvas". Se for definido como zero, será o maior tamanho possível.

Por padrão, ambos serão definidos como zero, o que tem o efeito de deixar o programa em tela-cheia.

Vamos precisar definir também variáveis globais que armazenarão o tamanho da janela e sua posição. Se estivermos rodando o jogo em um navegador, seus valores nunca mudarão, e serão os que forem indicados por tais macros. Mas se o jogo estiver rodando em uma janela, um usuário ou o próprio programa pode querer modificar seu tamanho.

Saber a altura e largura da janela em que estamos tem importância central para podermos desenhar na tela uma interface. Saber a posição da janela é muito menos útil. Entretanto, podemos pensar em conceitos experimentais de jogos que podem levar em conta tal informação. Talvez possa-se criar uma janela que tente evitar ser fechada movendo-se caso o mouse aproxime-se dela para fechá-la. Ou um jogo que crie uma janela que ao ser movida pela Área de trabalho possa revelar imagens diferentes, como se funcionasse como um raio-x da tela.

Além destas variáveis globais, será importante também criarmos um mutex a ser bloqueado sempre que elas forem modificadas em jogos com mais de uma thread:

**Seção: Cabeçalhos Weaver (continuação):**

```
extern int W_width, W_height, W_x, W_y;
#ifdef W_MULTITHREAD
extern pthread_mutex_t _window_mutex;
#endif
```

Estas variáveis precisarão ser atualizadas caso o tamanho da janela mude e caso a janela seja movida. E não são variáveis que o programador deva mudar. Não atribua nada à elas, são variáveis somente para leitura.

## 3.1 - Criar janelas

O código de criar janelas só será usado se estivermos compilando um programa nativo. Por isso, só iremos definir e declarar suas funções se a macro `W_TARGET` for igual à `W_ELF` .

**Seção: Cabeçalhos Weaver:**

```
#if W_TARGET == W_ELF
#include "window.h"
#endif
```

E o cabeçalho em si terá a forma:

**Arquivo:** `project/src/weaver/window.h`**:**

```
#ifndef _window_h_
#define _window_h_
#ifdef __cplusplus
  extern "C" {
#endif
```

<**Seção a ser Inserida: Inclui Cabeçalho de Configuração**>

```
#include "weaver.h"
#include "memory.h"
#include <signal.h>
#include <stdio.h> // fprintf
#include <stdlib.h> // exit
#include <X11/Xlib.h> // XOpenDisplay, XCloseDisplay, DefaultScreen,
                      // DisplayPlanes, XFree, XCreateSimpleWindow,
                      // XDestroyWindow, XChangeWindowAttributes,
                      // XSelectInput, XMapWindow, XNextEvent,
                      // XSetInputFocus, XStoreName,
#include <GL/gl.h>
#include <GL/glx.h> // glXChooseVisual, glXCreateContext, glXMakeCurrent
#include <X11/extensions/Xrandr.h> // XRRSizes, XRRRates, XRRGetScreenInfo,
                                   // XRRConfigCurrentRate,
                                   // XRRConfigCurrentConfiguration,
                                   // XRRFreeScreenConfigInfo,
                                   // XRRSetScreenConfigAndRate
#include <X11/XKBlib.h> // XkbKeycodeToKeysym
void _initialize_window(void);
void _finalize_window(void);
```

<**Seção a ser Inserida: Janela: Declaração**>

```
#ifdef __cplusplus
  }
#endif
#endif
```

Enquanto o próprio arquivo de definição de funções as definirá apenas condicionalmente:

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

<**Seção a ser Inserida: Inclui Cabeçalho de Configuração**>

```
  // Se W_TARGET != W_ELF, então este arquivo não terá conteúdo nenhum
  // para o compilador, o que é proibido pelo padrão ISO. A variável a
  // seguir que nunca será usada e nem declarada propriamente previne
  // isso.
extern int make_iso_compilers_happy;
#if W_TARGET == W_ELF
#include "window.h"
  int W_width, W_height, W_x, W_y;
#ifdef W_MULTITHREAD
  pthread_mutex_t _window_mutex;
```

```
#endif
                    <Seção a ser Inserida: Variáveis de Janela>
void _initialize_window(void){
                    <Seção a ser Inserida: Janela: Inicialização>
}
void _finalize_window(void){
                    <Seção a ser Inserida: Janela: Pré-Finalização>
                    <Seção a ser Inserida: Janela: Finalização>
}
                    <Seção a ser Inserida: Janela: Definição>
#endif
```

Desta forma, nada disso será incluído desnecessariamente quando compilarmos para a Web. Mas caso seja incluso, precisamos invocar uma função de inicialização e finalização na inicialização e finalização da API:

**Seção: API Weaver: Inicialização (continuação):**

```
#ifdef W_MULTITHREAD
  if(pthread_mutex_init(&_window_mutex, NULL) != 0){ // Inicializa mutex
    perror(NULL);
    exit(1);
}
#endif
#if W_TARGET == W_ELF
_initialize_window();
#endif
```

**Seção: API Weaver: Finalização (continuação):**

```
#ifdef W_MULTITHREAD
  if(pthread_mutex_destroy(&_window_mutex) != 0){ // Finaliza mutex
    perror(NULL);
    exit(1);
}
#endif
#if W_TARGET == W_ELF
_finalize_window();
#endif
```

Para que possamos criar uma janela, como o Xlib funciona segundo um modelo cliente-servidor, precisaremos de uma conexão com tal servidor. Tipicamente, tal conexão é chamada de "Display". Na verdade, além de ser uma conexão, um Display também armazena informações sobre o servidor com o qual nos conectamos. Como ter acesso à conexão é necessário para fazer muitas coisas diferentes, tais como obter entrada e saída, teremos que definir o nosso display como variável global para que esteja acessível para outros módulos.

**Seção: Variáveis de Janela:**

```
Display *_dpy;
```

**Seção: Cabeçalhos Weaver (continuação):**

```
#if W_TARGET == W_ELF
#include <X11/Xlib.h>
  extern Display *_dpy;
#endif
```

Ao inicializar uma conexão, o que pode dar errado é que podemos fracassar, talvez por o servidor não estar ativo. Como iremos abrir uma conexão com o servidor na própria máquina em que estamos executando, então não é necessário passar qualquer argumento para a função `XOpenDisplay` :

**Seção: Janela: Inicialização:**

```
  _dpy = XOpenDisplay(NULL);
  if(_dpy == NULL){
    fprintf(stderr,
            "ERROR: Couldn't connect with the X Server. Are you running a "
            "graphical interface?\n");
    exit(1);
  }
```

Nosso próximo passo será obter o número da tela na qual a janela estará. Teoricamente um dispositivo pode ter várias telas diferentes. Na prática provavelmente só encontraremos uma. Caso uma pessoa tenha duas ou mais, ela provavelmente ativa a extensão Xinerama, que faz com que suas duas telas sejam tratadas como uma só (tipicamente com uma largura bem grande). De qualquer forma, obter o ID desta tela será importante para obtermos alguns dados como a resolução máxima e quantidade de bits usado em cores.

**Seção: Variáveis de Janela (continuação):**

```
static int screen;
```

Para inicializar o valor, usamos a seguinte macro, a qual nunca falhará:

**Seção: Janela: Inicialização:**

```
  screen = DefaultScreen(_dpy);
```

Como a tela é um inteiro, não há nada que precisemos desalocar depois. E de posse do ID da tela, podemos obter algumas informações à mais como a profundidade dela. Ou seja, quantos bits são usados para representar as cores.

**Seção: Variáveis de Janela (continuação):**

```
static int depth;
```

No momento da escrita deste texto, o valor típico da profundidade de bits é de 24. Assim, as cores vermelho, verde e azul ficam cada uma com 8 bits (totalizando 24) e 8 bits restantes ficam representando um valor alpha que armazena informação de transparência.

**Seção: Janela: Inicialização (continuação):**

```
  depth = DisplayPlanes(_dpy, screen);
  #if W_DEBUG_LEVEL >= 3
  printf("WARNING (3): Color depth: %d\n", depth);
  #endif
```

De posse destas informaões, já podemos criar a nossa janela. Ela é declarada assim:

**Seção: Variáveis de Janela:**

```
Window _window;
```

**Seção: Cabeçalhos Weaver (continuação):**

```
#if W_TARGET == W_ELF
#include <X11/Xlib.h>
  extern Window _window;
#endif
```

E é inicializada com os seguintes dados:

**Seção: Janela: Inicialização (continuação):**

```
  W_x = 0; // Na inicialização não é necessário ativar o mutex.
  W_y = 0;
```

```
#if W_WIDTH > 0
  W_width = W_WIDTH; // Obtendo largura da janela
#else
  W_width = DisplayWidth(_dpy, screen);
#endif
#if W_HEIGHT > 0 // Obtendo altura da janela
  W_height = W_HEIGHT;
#else
  W_height = DisplayHeight(_dpy, screen);
#endif
  _window = XCreateSimpleWindow(_dpy, //Conexão com o servidor X
                                DefaultRootWindow(_dpy), // A janeela-mãe
                                W_x, W_y, // Coordenadas da janela
                                W_width, // Largura da janela
                                W_height, // Altura da janela
                                0, 0, // Borda (espessura e cor)
                                0); // Cor padrão
```

Isso cria a janela. Mas isso não quer dizer que a janela será exibida. Ainda temos que fazer algumas coisas como mudar alguns atributos da sua configuração. Só depois disso poderemos pedir para que o servidor mostre a janela visualmente.

Vamos nos concentrar agora nos atributos da janela. Primeiro nós queremos que nossas escolhas de configuração sejam as mais soberanas possíveis. Devemos pedir que o gerenciador de janelas faça todo o possível para cumpri-las. Por isso, começamos ajustando a flag "Override Redirect", o que propagandeia nossa janela como uma janela de"pop-up". Isso faz com que nossos pedidos de entrar em tela cheia sejam atendidos, mesmo quando estamos em ambientes como o XMonad.

A próxima coisa que fazemos é informar quais eventos devem ser notificados para nossa janela. No caso, queremos ser avisados quando um botão é pressionado, liberado, bem como botões do mouse e quando a janela é revelada ou tem o seu tamanho mudado.

E por fim, mudamos tais atributos na janela e fazemos o pedido para começarmos a ser notificados de quando houverem eventos de entrada:

**Seção: Variáveis de Janela (continuação):**

```
static XSetWindowAttributes at;
```

**Seção: Janela: Inicialização (continuação):**

```
  {
    at.override_redirect = True;
    // Eventos que nos interessam: pressionar e soltar botão do
    // teclado, pressionar e soltar botão do mouse, movimento do mouse,
    // quando a janela é exposta e quando ela muda de tamanho.
    at.event_mask = ButtonPressMask | ButtonReleaseMask | KeyPressMask |
      KeyReleaseMask | PointerMotionMask | ExposureMask | StructureNotifyMask;
    XChangeWindowAttributes(_dpy, _window, CWOverrideRedirect, &at);
    XSelectInput(_dpy, _window, StructureNotifyMask | KeyPressMask |
                 KeyReleaseMask | ButtonPressMask | ButtonReleaseMask |
                 PointerMotionMask | ExposureMask | StructureNotifyMask);
  }
```

Agora o que enfim podemos fazer é pedir para que a janela seja desenhada na tela. Primeiro pedimos sua criação e depois aguardamos o evento de sua criação. Quando formos notificados do evento, pedimos para que a janela receba foco, mas que devolva o foco para a janela-mãe quando

terminar de executar. Ajustamos o nome que aparecerá na barra de título do programa. E se nosso programa tiver várias threads, avisamos o Xlib disso:

**Seção: Janela: Inicialização (continuação):**

```
  XMapWindow(_dpy, _window);
  {
    XEvent e;
    XNextEvent(_dpy, &e);
    while(e.type != MapNotify){
      XNextEvent(_dpy, &e);
    }
  }
  XSetInputFocus(_dpy, _window, RevertToParent, CurrentTime);
#ifdef W_PROGRAM_NAME
  XStoreName(_dpy, _window, W_PROGRAM_NAME);
#else
  XStoreName(_dpy, _window, W_PROG);
#endif
#ifdef W_MULTITHREAD
  XInitThreads();
#endif
```

Antes de inicializarmos o código para OpenGL, precisamos garantir que tenhamos uma versão do GLX de pelo menos 1.3. Antes disso, não poderíamos ajustar as configurações do contexto OpenGL como queremos. Sendo assim, primeiro precisamos checar se estamos com uma versão compatível:

**Seção: Janela: Inicialização (continuação):**

```
{
  int glx_major, glx_minor;
  Bool ret;
  ret = glXQueryVersion(_dpy, &glx_major, &glx_minor);
  if(!ret || (( glx_major == 1 ) && ( glx_minor < 3 )) || glx_major < 1){
    fprintf(stderr,
            "ERROR: GLX is version %d.%d, but should be at least 1.3.\n",
            glx_major, glx_minor);
    exit(1);
  }
}
```

A última coisa que precisamos fazer agora na inicialização é criar um contexto OpenGL e associá-lo à nossa recém-criada janela para que possamos usar OpenGL nela:

**Seção: Variáveis de Janela:**

```
  static GLXContext context;
```

Também vamos precisar de configurações válidas para o nosso contexto:

**Seção: Variáveis de Janela:**

```
  static GLXFBConfig *fbConfigs;
```

Estas são as configurações que queremos para termos uma janela colorida que pode ser desenhada e com buffer duplo.

**Seção: Janela: Inicialização (continuação):**

```
{
  int return_value;
```

```
  int doubleBufferAttributes[] = {
    GLX_DRAWABLE_TYPE, GLX_WINDOW_BIT, // Desenharemos na tela, não em 'pixmap'
    GLX_RENDER_TYPE,   GLX_RGBA_BIT, // Definimos as cores via RGBA, não paleta
    GLX_DOUBLEBUFFER,  True, // Usamos buffers duplos para evitar 'flickering'
    GLX_RED_SIZE,      1, // Devemos ter ao menso 1 bit de vermelho
    GLX_GREEN_SIZE,    1, // Ao menos 1 bit de verde
    GLX_BLUE_SIZE,     1, // Ao menos 1 bit de azul
    GLX_ALPHA_SIZE,    1, // Ao menos 1 bit para o canal alfa
    GLX_DEPTH_SIZE,    1, // E ao menos 1 bit de profundidade
    None
  };
  fbConfigs = glXChooseFBConfig(_dpy, screen, doubleBufferAttributes,
                                &return_value);
  if (fbConfigs == NULL){
    fprintf(stderr,
          "ERROR: Not possible to choose our minimal OpenGL configuration.\n");
    exit(1);
  }
}
```

Agora iremos precisar usar uma função chamada `glXCreateContextAttribsARB` para criar um contexto OpenGL 3.0. O problema é que nem todas as placas de vídeo possuem ela. Algumas podem não ter suporte à versões mais novas do openGL. Por causa disso, a API não sabe se esta função existe ou não e ela não está sequer declarada. Nós mesmos precisamos declará-la e obter o seu valor dinamicamente verificando se ela existe:

**Seção: Janela: Declaração (continuação):**

```
typedef GLXContext
  (*glXCreateContextAttribsARBProc)(Display*, GLXFBConfig, GLXContext, Bool,
                                    const int*);
```

Tendo declarado o novo tipo, tentamos obter a função e usá-la para criar o contexto:.

**Seção: Janela: Inicialização (continuação):**

```
{
  int context_attribs[] =
    { //  Iremos usar e exigir OpenGL 3.3
      GLX_CONTEXT_MAJOR_VERSION_ARB, 3,
      GLX_CONTEXT_MINOR_VERSION_ARB, 3,
      None
    };
  glXCreateContextAttribsARBProc glXCreateContextAttribsARB = 0;
  { // Verificando se a 'glXCreateContextAttribsARB' existe:
    // Usamox 'glXQueryExtensionsString' para obter lista de extensões
    const char *glxExts = glXQueryExtensionsString(_dpy, screen);
    if(strstr(glxExts, "GLX_ARB_create_context") == NULL){
      fprintf(stderr, "ERROR: Can't create an OpenGL 3.0 context.\n");
      exit(1);
    }
  }
  // Se estamos aqui, a função existe. Obtemos seu endereço e a usamos
  // para criar o contexto OpenGL.
  glXCreateContextAttribsARB = (glXCreateContextAttribsARBProc)
```

```
    glXGetProcAddressARB( (const GLubyte *) "glXCreateContextAttribsARB" );
  context = glXCreateContextAttribsARB(_dpy, *fbConfigs, NULL, GL_TRUE,
                                       context_attribs);
  glXMakeCurrent(_dpy, _window, context);
}
```

À partir de agora, se tudo deu certo e suportamos todos os pré-requisitos, já criamos a nossa janela e ela está pronta para receber comandos OpenGL. Agora é só na finalização destruirmos o contexto que criamos. Colocamos logo em seguida o código para destruir a janela e encerrar a conexão, já que estas coisas precisam ser feitas nesta ordem:

**Seção: Janela: Finalização (continuação):**

```
  glXMakeCurrent(_dpy, None, NULL);
  glXDestroyContext(_dpy, context);
  XDestroyWindow(_dpy, _window);
  XCloseDisplay(_dpy);
```

# 3.2 - Definir tamanho do canvas

Agora é hora de definirmos também o espaço na qual poderemos desenhar na tela quando compilamos o programa para a Web. Felizmente, isso é mais fácil que criar uma janela no Xlib. Basta usarmos o suporte que Emscripten tem para as funções SDL. Então adicionamos como cabeçalho da API:

**Seção: Cabeçalhos Weaver:**

```
#if W_TARGET == W_WEB
#include "canvas.h"
#endif
```

Agora definimos o nosso cabeçalho do módulo de "canvas":

**Arquivo:** `project/src/weaver/canvas.h`**:**

```
#ifndef _canvas_H_
#define _canvas_h_
#ifdef __cplusplus
  extern "C" {
#endif
            <Seção a ser Inserida: Inclui Cabeçalho de Configuração>
#include "weaver.h"
#include <stdio.h> // |fprintf|
#include <stdlib.h> // |exit|
#include <SDL/SDL.h> // |SDL_Init|, |SDL_CreateWindow|, |SDL_DestroyWindow|,
                     // |SDL_Quit|
void _initialize_canvas(void);
void _finalize_canvas(void);
                    <Seção a ser Inserida: Canvas: Declaração>
#ifdef __cplusplus
  }
#endif
#endif
```

E por fim, o nosso `canvas.c` que definirá as funções que criarão nosso espaço de desenho pode ser definido. Como ele é bem mais simples, será inteiramente definido abaixo:

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

```
            <Seção a ser Inserida: Inclui Cabeçalho de Configuração>
```

```
extern int make_iso_compilers_happy;
#if W_TARGET == W_WEB
#include "canvas.h"
static SDL_Surface *window;
int W_width, W_height, W_x = 0, W_y = 0;
#ifdef W_MULTITHREAD
pthread_mutex_t _window_mutex;
#endif
```

<Seção a ser Inserida: Canvas: Variáveis>

```
void _initialize_canvas(void){
  SDL_Init(SDL_INIT_VIDEO); // Inicializando SDL com OpenGL 3.3
  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
  SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
  window = SDL_SetVideoMode(// Definindo informações de tamanho do canvas
#if W_WIDTH > 0
                            W_width = W_WIDTH, // Largura da janela
#else
                            W_width = 800, // Largura da janela
#endif
#if W_HEIGHT > 0
                            W_height = W_HEIGHT, // Altura da janela
#else
                            W_height = 600, // Altura da janela
#endif
                            0, // Bits por pixel, usar o padrão
                            SDL_OPENGL // Inicializar o contexto OpenGL
#if W_WIDTH == 0 && W_HEIGHT == 0
                            | SDL_WINDOW_FULLSCREEN
#endif
                            );
  if (window == NULL) {
    fprintf(stderr, "ERROR: Could not create window: %s\n", SDL_GetError());
    exit(1);
  }
```

<Seção a ser Inserida: Canvas: Inicialização>

```
}
  void _finalize_canvas(void){// Desalocando a nossa superfície de canvas
  SDL_FreeSurface(window);
}
```

<Seção a ser Inserida: Canvas: Definição>

```
#endif
```

Note que o que estamos chamando de ”janela” na verdade é uma superfície SDL. E que não é necessário chamar `SDL_Quit` , tal função seria ignorada se usada.

Por fim, basta agora apenas invocarmos tais funções na inicialização e finalização da API:

**Seção: API Weaver: Inicialização (continuação):**

```
#if W_TARGET == W_WEB
_initialize_canvas();
#endif
```

**Seção: API Weaver: Finalização (continuação):**

```
#if W_TARGET == W_WEB
  _finalize_canvas();
#endif
```

## 3.3 - Mudanças no Tamanho e Posição da Janela

Em Xlib, quando uma janela tem o seu tamanho mudado, ela recebe um evento do tipo `ConfigureNotify` . Além dele, também existirão novos eventos se o usuário apertar uma tecla, mover o mouse e assim por diante. Por isso, precisamos adicionar código para tratarmos de eventos no loop principal:

**Seção: API Weaver: Loop Principal:**

```
    <Seção a ser Inserida: API Weaver: Imediatamente antes de tratar eventos>
#if W_TARGET == W_ELF
  {
    XEvent event;
    while(XPending(_dpy)){
      XNextEvent(_dpy, &event);
      // A variável 'event' terá todas as informações do evento
                <Seção a ser Inserida: API Weaver: Trata Evento Xlib>
    }
  }
#endif
#if W_TARGET == W_WEB
  {
    SDL_Event event;
    while(SDL_PollEvent(&event)){
      // A variável 'event' terá todas as informações do evento
                <Seção a ser Inserida: API Weaver: Trata Evento SDL>
    }
  }
#endif
```

Por hora definiremos só o tratamento do evento de mudança de tamanho e posição da janela em Xlib. Outros eventos terão seus tratamentos definidos mais tarde, assim como os eventos SDL caso estejamos rodando em um navegador web.

Tudo o que temos que fazer no caso deste evento é atualizar as variáveis globais `W_width` , `W_height` , `W_x` e `W_y` . Nem sempre o evento `ConfigureNotify` significa que a janela mudou de tamanho ou foi movida. Talvez ela apenas tenha se movido para frente ou para trás em relação à outras janelas empilhadas sobre ela. Ou algo mudou o tamanho de sua borda. Mas mesmo assim, não custa quase nada atualizarmos tais dados. Se eles não mudaram, de qualquer forma o código será inócuo:

**Seção: API Weaver: Trata Evento Xlib:**

```
if(event.type == ConfigureNotify){
  XConfigureEvent config = event.xconfigure;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&_window_mutex);
#endif
  W_x = config.x;
  W_y = config.y;
  W_width = config.width;
```

```
  W_height = config.height;
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_window_mutex);
#endif
  continue;
}
```

Não é necessário criar um código análogo para a Web, pois lá será impossível mover a nossa "janela", que será um *canvas*.

Mas e se nós quisermos mudar o tamanho ou a posição de uma janela diretamente? Para mudar o tamanho, precisamos definir separadamente o código tanto para o caso de termos uma janela como para o caso de termos um *canvas* web para o jogo. No caso da janela, usamos uma função XLib para isso:

**Seção: Janela: Declaração:**

```
  void Wresize_window(int width, int height);
```

**Seção: Janela: Definição:**

```
void Wresize_window(int width, int height){
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&_window_mutex);
#endif
  XResizeWindow(_dpy, _window, width, height);
  W_width = width;
  W_height = height;
            <Seção a ser Inserida: Ações após Redimencionar Janela>
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_window_mutex);
#endif
}
```

No caso de termos um "canvas" web, então usamos SDL para obtermos o mesmo efeito. Basta pedirmos para criar uma nova janela e isso funciona como se mudássemos o tamanho da anterior:

**Seção: Canvas: Declaração:**

```
  void Wresize_window(int width, int height);
```

**Seção: Canvas: Definição:**

```
void Wresize_window(int width, int height){
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&_window_mutex);
#endif
  window = SDL_SetVideoMode(width, height,
                            0, // Bits por pixel, usar o padrão
                            SDL_OPENGL // Inicializar o contexto OpenGL
                            );
  W_width = width;
  W_height = height;
            <Seção a ser Inserida: Ações após Redimencionar Janela>
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_window_mutex);
#endif
}
```

Mudar a posição da janela é algo diferente. Isso só faz sentido se realmente tivermos uma janela Xlib, e não um "canvas" web. De qualquer forma, precisaremos definir esta função em ambos os casos.

**Seção: Janela: Declaração:**

```
  void Wmove_window(int x, int y);
```

**Seção: Janela: Definição:**

```
void Wmove_window(int x, int y){
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&_window_mutex);
#endif
  XMoveWindow(_dpy, _window, x, y);
  W_x = x;
  W_y = y;
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_window_mutex);
#endif
}
```

Esta mesma função será definida, mas será ignorada se um usuário a invocar em um programa compilado para a Web:

**Seção: Canvas: Declaração:**

```
  void Wmove_window(int x, int y);
```

**Seção: Canvas: Definição:**

```
void Wmove_window(int width, int height){
  return;
}
```

# 3.4 - Mudando a resolução da tela

Inicialmente o Servidor X não possuía qualquer recurso para que fosse possível mudar a sua resolução enquanto ele executa, ou coisas como rotacionar a janela raiz. A única forma de obter isso era encerrando o servidor e iniciando-o novamente com nova configuração. Mas programas como jogos podem ter a necessidade de rodar em resolução menor para melhorar o desempenho, mas ao mesmo tempo podem precisar ocupar a tela toda para obter imersão.

Note que isso só faz sentido quando lidamos com uma janela rodando em um gerenciador de janelas. Não no *canvas* de um navegador.

O primeiro problema que temos é que não dá pra mudar a resolução arbitrariamente. Existe apenas um conjunto limitado de resoluções que são realmente possíveis em um dado monitor. Então a primeira coisa que precisamos fazer é descobrir quantos modos são realmente possíveis na tela em que estamos.

Cada modo de funcionamento suportado por uma tela possui três valores distintos: a resolução horizontal, vertical, e a frequência de atualização da tela. A ideia é que nós usemos uma variável `Wnumber_of_modes` para armazenar quantos modos diferentes temos, `Wcurrent_mode` para sabermos qual o modo atual e aloquemos uma estrutura formada por um *array* de triplas de números contendo os dados de cada modo, a qual pode ser acessada por meio de `Wmodes` . Cada um dos modos possíveis terá um número sequencial. E se quisermos passar para outro modo, usaremos uma função que recebe como argumento tal número e facilmente pode checar se está diante de um valor inválido.

**Seção: Variáveis de Janela (continuação):**

```
  unsigned Wnumber_of_modes, Wcurrent_mode;
  struct _wmodes{
```

```
    int width, height, rate, id;
  } *Wmodes;
```

**Seção: Canvas: Variáveis:**

```
  unsigned Wnumber_of_modes, Wcurrent_mode;
  struct _wmodes{
    int width, height, rate, id;
  } *Wmodes;
```

**Seção: Cabeçalhos Weaver (continuação):**

```
  extern unsigned Wnumber_of_modes, Wcurrent_mode;
  extern struct _wmodes *Wmodes;
```

Agora cabe à nós inicializarmos isso tudo. Se estamos programando para a Web, nós não podemos mesmo mudar a resolução. Então, o número de modos que temos é sempre um só. E a informação de resolução de tela pode ser obtida armazenando o retorno de `SDL_GetVideoInfo` em uma estrutura de informação. A taxa de atualização de tela é setada como zero, significando um valor indefinido.

**Seção: Canvas: Inicialização:**

```
  {
    const SDL_VideoInfo *info = SDL_GetVideoInfo();
    Wnumber_of_modes = 1;
    Wcurrent_mode = 0;
    Wmodes = (struct _wmodes *) _iWalloc(sizeof(struct _wmodes));
    Wmodes[0].width = info->current_w;
    Wmodes[0].height = info->current_h;
    Wmodes[0].rate = 0;
    Wmodes[0].id = 0;
  }
#if W_DEBUG_LEVEL >=3
  fprintf(stderr, "WARNING (3): Screen resolution: %dx%d.\n",
          Wmodes[0].width, Wmodes[0].height);
#endif
```

Se não estamos programando para a Web, inicializar tais dados é mais complicado. Nós vamos precisar usar a extensão XRandr. E além disso, como podemos mudar a resolução da nossa tela, é importante memorizarmos os valores iniciais para podermos restaurá-los antes de terminar o programa. Tanto quando o programa encerra naturalmente como quando é encerrado à força por meio de uma falha de segmentação. Usaremos um punhado de variáveis para armazenar os dadso que precisamos para isso.

A primeira é `_orig_size_id` , um ID que representa a resolução e a segunta é `_orig_rate` , que representa a taxa de atualização a tela. Mais abaixo, `_orig_rotation` armazena a rotação atual da tela. Weaver não permite que rotacionemos a tela, mas mesmo assim tal informação deve ser obtida para quando depois tivermos que restaurar as configurações iniciais.

Por fim, a quarta variável que definimos é uma que irá armazenar as informações das configurações relacionadas à resolução e taxa de atualização da tela.

**Seção: Variáveis de Janela (continuação):**

```
  static int _orig_size_id, _orig_rate;
  static Rotation _orig_rotation;
  static XRRScreenConfiguration *conf;
```

**Seção: Janela: Inicialização (continuação):**

```
{
  Window root = RootWindow(_dpy, 0); // Janela raíz da tela padrão
```

```
  int num_modes, num_rates, i, j, k;
  // Obtendo uma lista de todas as resoluções possíveis:
  XRRScreenSize *modes = XRRSizes(_dpy, 0, &num_modes);
  short *rates;
  // Obtendo lista de taxa de atualização do monitor em cada resolução
  // e com isso concluindo o número total de modos diferentes:
  Wnumber_of_modes = 0;
  for(i = 0; i < num_modes; i ++){
    rates = XRRRates(_dpy, 0, i, &num_rates);
    Wnumber_of_modes += num_rates;
  }
  // Alocamos na arena de memória interna espaço para contermos dados
  // sobre todas as combinações possíveis de resolução e taxa de
  // atualização:
  Wmodes = (struct _wmodes *) _iWalloc(sizeof(struct _wmodes) *
                                       Wnumber_of_modes);
  // Obtendo o valor original de resolução e taxa de atualização:
  conf = XRRGetScreenInfo(_dpy, root);
  _orig_rate = XRRConfigCurrentRate(conf);
  _orig_size_id = XRRConfigCurrentConfiguration(conf, &_orig_rotation);
  // Preenchendo as informações dos modos e descobrindo o ID do modo atual
  k = 0;
  for(i = 0; i < num_modes; i ++){
    rates = XRRRates(_dpy, 0, i, &num_rates);
    for(j = 0; j < num_rates; j++){
      Wmodes[k].width = modes[i].width;
      Wmodes[k].height = modes[i].height;
      Wmodes[k].rate = rates[j];
      Wmodes[k].id = i;
      if(i == _orig_size_id && rates[j] == _orig_rate)
        Wcurrent_mode = k;
      k ++;
    }
  }
#if W_DEBUG_LEVEL >=3
  fprintf(stderr, "WARNING (3): Screen resolution: %dx%d (%dHz).\n",
          Wmodes[Wcurrent_mode].width, Wmodes[Wcurrent_mode].height,
          Wmodes[Wcurrent_mode].rate);
#endif
}
```

Caso modifiquemos a resolução da tela, antes de fechar o programa, precisamos fazer tudo voltar ao que era antes. Mesmo se o programa for encerrado devido à uma falha de segmentação, divisão por zero, ou algo assim. Independente do que causar o fim do programa, precisamos chamar a função que definiremos:

**Seção: Janela: Declaração:**

```
  void _restore_resolution(void);
```

**Seção: Janela: Definição:**

```
void _restore_resolution(void){
#ifdef W_MULTITHREAD
```

```
  pthread_mutex_lock(&_window_mutex);
#endif
  Window root = RootWindow(_dpy, 0);
  XRRSetScreenConfigAndRate(_dpy, conf, root, _orig_size_id, _orig_rotation,
                            _orig_rate, CurrentTime);
  XRRFreeScreenConfigInfo(conf);
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_window_mutex);
#endif

}
```

O primeiro caso no qual chamamos esta função é quando encerramos o programa normalmente. Mas precisamos chamar ela antes de termos fechado a conexão com o servidor X. Por isso colocamos este código de finalização imediatamente antes:

**Seção: Janela: Pré-Finalização:**

```
  _restore_resolution();
```

Mas e se o programa tiver que ser encerrado devido à algum sinal fatal? Pode ter ocorrido uma falha de segmentação ou uma divisão por zero. Neste caso, precisamos restaurar a resolução antes de encerrar o programa. Para isso temos que substituir a ação padrão em cada sinal letal por uma função que faz isso:

**Seção: Cabeçalhos Weaver:**

```
#if W_TARGET == W_ELF
#include <signal.h>
#endif
```

**Seção: API Weaver: Inicialização (continuação):**

```
#if W_TARGET == W_ELF
{
  struct sigaction sa;
  memset(&sa, 0, sizeof(struct sigaction));
  sigemptyset(&sa.sa_mask);
  // _restore_and_quit é uma função que definiremos logo em
  // seguida. Ela deverá ser executada ao invés da função normal
  // associada a cada sinal:
  sa.sa_sigaction = _restore_and_quit;
  sa.sa_flags   = SA_SIGINFO;
  sigaction(SIGHUP, &sa, NULL);
  sigaction(SIGINT, &sa, NULL);
  sigaction(SIGQUIT, &sa, NULL);
  sigaction(SIGILL, &sa, NULL);
  sigaction(SIGABRT, &sa, NULL);
  sigaction(SIGFPE, &sa, NULL);
  sigaction(SIGSEGV, &sa, NULL);
  sigaction(SIGPIPE, &sa, NULL);
  sigaction(SIGALRM, &sa, NULL);
  sigaction(SIGTERM, &sa, NULL);
  sigaction(SIGUSR1, &sa, NULL);
  sigaction(SIGUSR2, &sa, NULL);
}
#endif
```

Como indicado pelo código, ao invés da função normal, o que iremos executar sempre que recebermos um sinal fatal será a função abaixo:

**Seção: Janela: Declaração:**

```
  void _restore_and_quit(int signal, siginfo_t *si, void *arg);
```

**Seção: Janela: Definição:**

```
void _restore_and_quit(int signal, siginfo_t *si, void *arg){
  _restore_resolution();
  // Restaura todos os sinais salvos:
  switch(signal){
  case SIGHUP:
    fprintf(stderr, "Program hangup.\n");
    break;
  case SIGINT:
  case SIGTERM:
    fprintf(stderr, "Program terminated.\n");
    break;
  case SIGQUIT:
    fprintf(stderr, "Program quited.\n");
    break;
  case SIGILL:
    fprintf(stderr, "Ilegal instruction.\n");
    break;
  case SIGABRT:
    fprintf(stderr, "Program aborted.\n");
    break;
  case SIGFPE:
    fprintf(stderr, "Erroneous arithmetic expression (divided by zero?).\n");
    break;
  case SIGSEGV:
    fprintf(stderr, "Segmentation fault.\n");
    break;
  case SIGPIPE:
    fprintf(stderr, "Broken pipe.\n");
    break;
  case SIGALRM:
    fprintf(stderr, "Program terminated by alarm clock.\n");
    break;
  default:
    fprintf(stderr, "Program terminated by unknown signal.\n");
  }
  exit(1);
  // Este código nunca será executado, mas previne aviso de compilação
  // por não usarmos os argumentos 'si' e 'arg':
  *((int *) arg) = *((int *) si);
}
```

O único caso no qual não seremos capazes de restaurar a resolução é quando recebermos um `SIGKILL` . Não há muito a fazer com relação à isso. Entretanto, um sinal desta magnitude só pode ser gerado por um usuário, nunca será a reação do Sistema Operacional à uma ação do programa.

Teremos que assumir que caso isso aconteça, o usuário sabe o que está fazendo e saberá retornar a resolução ao seu estado atual.

Uma vez que tenhamos garantido que a resolução voltará ao normal após o programa se encerrar, podemos fornecer então uma função responsável por mudar a resolução e modo da tela. Esta função deverá receber como argumento um número inteiro. Se este número for menor que zero ou maior ou igual ao número total de modos que temos em nossa tela, a função não fará nada e retornará zero. Caso contrário, ela mudará o modo da tela para o representado pelo índice passado como argumento em `Wmodes`. Além disso, ela mudará o tamanho da janela para o da nova resolução, deixando o jogo em tela cheia, e retornará 1:

**Seção: Janela: Declaração:**

```
int Wfullscreen_mode(unsigned int mode);
```

**Seção: Janela: Definição:**

```
int Wfullscreen_mode(unsigned int mode){
  if(mode >= Wnumber_of_modes)
    return 0;
  else{
    Window root = RootWindow(_dpy, 0);
    Wmove_window(0, 0);
    Wresize_window(Wmodes[mode].width, Wmodes[mode].height);
    XRRSetScreenConfigAndRate(_dpy, conf, root, Wmodes[mode].id, _orig_rotation,
                              Wmodes[mode].rate, CurrentTime);
    return 1;
  }
}
```

Também teremos que definir a mesma função caso estejamos fazendo um jogo para a Web. Mas neste caso, a função não fará sentido e sempre retornará 0:

**Seção: Canvas: Declaração:**

```
int Wfullscreen_mode(int mode);
```

**Seção: Canvas: Definição:**

```
int Wfullscreen_mode(int mode){
  return 0;
}
```

# 3.5 - Configurações Básicas OpenGL

A única configuração que temos no momento é a cor de fundo de nossa janela, a qual será exibida na ausência de qualquer coisa a ser mostrada:

**Seção: API Weaver: Inicialização (continuação):**

```
// Com que cor limpamos a tela:
glClearColor(W_DEFAULT_COLOR, 1.0f);
// Ativamos o buffer de profundidade:
glEnable(GL_DEPTH_TEST);
```

**Seção: API Weaver: Loop Principal (continuação):**

```
glClear(GL_COLOR_BUFFER_BIT);
```

# Capítulo 4: Teclado e Mouse

Uma vez que tenhamos uma janela, podemos começar a acompanhar os eventos associados à ela. Um usuário pode apertar qualquer botão no seu teclado ou mouse e isso gerará um evento. Devemos tratar tais eventos no mesmo local em que já estamos tratando coisas como o mover e o mudar tamanho da janela (algo que também é um evento). Mas devemos criar uma interface mais simples para que um usuário possa acompanhar quando certas teclas são pressionadas, quando são soltas, por quanto tempo elas estão sendo pressionadas e por quanto tempo foram pressionadas antes de terem sido soltas.

Nossa proposta é que exista um vetor de inteiros chamado `Wkeyboard` , por exemplo, e que cada posição dele represente uma tecla diferente. Se o valor dentro de uma posição do vetor é 0, então tal tecla não está sendo pressionada. Caso o seu valor seja um número positivo, então a tecla está sendo pressionada e o número representa por quantos milissegundos a tecla vem sendo pressionada. Caso o valor seja um número negativo, significa que a tecla acabou de ser solta e o inverso deste número representa por quantos milissegundos a tecla ficou pressionada. E caso o valor seja 1, isso significa que a tecla começou a ser pressionada exatamente neste *frame*.

Acompanhar o tempo no qual uma tecla é pressionada é tão importante quanto saber se ela está sendo pressionada ou não. Por meio do tempo, podemos ser capazes de programar personagens que pulam mais alto ou mais baixo, dependendo do quanto um jogador apertou uma tecla, ou fazer com que jogadores possam escolher entre dar um soco rápido e fraco ou lento e forte em outros tipos de jogo. Tudo depende da intensidade com a qual eles pressionam os botões.

Entretanto, tanto o Xlib como SDL funcionam reportando apenas o momento no qual uma tecla é pressionada e o momento na qual ela é solta. Então, em cada iteração, precisamos memorizar quais teclas estão sendo pressionadas. Se duas pessoas estiverem compartilhando um mesmo teclado, teoricamente, o número máximo de teclas que podem ser pressionadas é 20 (se cada dedo da mão de cada uma delas estiver sobre uma tecla). Então, vamos usar um vetor de 20 posições para armazenar o número de cada tecla sendo pressionada. Isso é apenas para podermos atualizar em cada iteração do loop principal o tempo em que cada tecla é pressionada. Se hipoteticamente mais de 20 teclas forem pressionadas, o fato de perdermos uma delas não é algo muito grave e não deve causar qualquer problema.

Até agora estamos falando do teclado, mas o mesmo pode ser implementado nos botões do mouse. Mas no caso do mouse, além dos botões, temos o seu movimento. Então será importante armazenarmos a sua posição $(x, y)$, mas também um vetor representando a sua velocidade. Tal vetor deve considerar como se a posição atual do ponteiro do mouse fosse a $(0, 0)$ e deve conter qual a sua posição no próximo segundo caso o seu deslocamento continue constante na mesma direção e sentido em que vem sendo desde a última iteração. Desta forma, tal vetor também será útil para verificar se o mouse está em movimento ou não. E saber a intensidade e direção do movimento do mouse pode permitir interações mais ricas com o usuário.

## 4.1 - Preparando o Loop Principal: Medindo a Passagem de Tempo

Conforme exposto na introdução, toda vez que estivermos em um loop principal do jogo, a função `Wrest` deve ser invocada uma vez a cada iteração. Devemos então manter algumas variáveis controlando a passagem do tempo, e tais variáveis devem ser atualizadas sempre dentro destas funções.

No caso, vamos precisar inicialmente de uma variável para armazenar o tempo da iteração atual e a da iteração anterior, em escala de microssegundos:

**Seção: API Weaver: Definições:**

```
static struct timeval _last_time, _current_time;
```

É importante que ambos os valores sejam inicializados como zero, caso contrário, valores estranhos podem ser derivados caso usemos os valores antes de serem corretamente inicializados na primeira iteração de um loop principal:

**Seção: API Weaver: Inicialização (continuação):**

```
_last_time.tv_sec = 0;
_last_time.tv_sec = 0;
_current_time.tv_sec = 0;
_current_time.tv_usec = 0;
```

No loop principal em si, o valor que temos como o do tempo atual deve ser passado para o tempo anterior, e em seguida deve ser sobrescrito por um novo tempo atual:

**Seção: API Weaver: Loop Principal (continuação):**

```
{
  _last_time.tv_sec = _current_time.tv_sec;
  _last_time.tv_usec = _current_time.tv_usec;
  gettimeofday(&_current_time, NULL);
}
```

Estas medidas de tempo serão realmente usadas para atualizar duas variáveis a cada iteração. A primeira será uma variável interna e armazenará quantos milissegundos se passaram entre uma iteração e outra. A segunda será uma variável global que poderá ser consultada por usuários e conterá à quantos frames por segundo o jogo está rodando:

**Seção: API Weaver: Definições:**

```
  static int _elapsed_milisseconds;
  int Wfps;
```

**Seção: Cabeçalhos Weaver (continuação):**

```
    extern int Wfps;
```

Naturalmente, tais valores também precisam ser inicializados para prevenir que contenham números absurdos na primeira iteração:

**Seção: API Weaver: Inicialização (continuação):**

```
{
  _elapsed_milisseconds = 0;
  Wfps = 0;
}
```

E em cada iteração do loop principal, atualizamos os valores. Assim subtraímos dois `struct timeval` para obter os valores de `elapsed_milisseconds` e `Wfps` em cada *frame*:

**Seção: API Weaver: Loop Principal (continuação):**

```
{
  _elapsed_milisseconds = (_current_time.tv_sec - _last_time.tv_sec) * 1000;
  _elapsed_milisseconds += (_current_time.tv_usec - _last_time.tv_usec) / 1000;
  if(_elapsed_milisseconds > 0)
    Wfps = 1000 / _elapsed_milisseconds;
  else
    Wfps = 0;
}
```

## 4.2 - O Teclado

Para o teclado precisaremos de uma variável local que armazenará as teclas que já estão sendo pressionadas e uma variável global que será um vetor de números representando a quanto tempo cada tecla é pressionada. Adicionalmente, também precisamos tomar nota das teclas que acabaram

de ser soltas para que na iteração seguinte possamos zerar os seus valores no vetor de por quanto tempo estão pressionadas.

Mas a primeira questão que temos a responder é que tamanho deve ter tal vetor? E como associar cada posição à uma tecla?

Um teclado típico tem entre 80 e 100 teclas diferentes. Entretanto, diferentes teclados representam em cada uma destas teclas diferentes símbolos e caracteres. Alguns teclados possuem "Ç", outros possuem o símbolo do Euro, e outros podem possuir símbolos bem mais exóticos. Há também teclas modificadoras que transformam determinadas teclas em outras. O Xlib reconhece diferentes teclas associando à elas um número chamado de **KeySym**, que são inteiros de 29 bits.

Entretanto, não podemos criar um vetor de $2^{29}$ números para representar se uma das diferentes teclas possíveis está pressionada. Se cada inteiro tiver 4 bytes, vamos precisar de 2GB de memória para conter tal vetor. Por isso, precisamos nos ater à uma quantidade menor de símbolos.

A vasta maioria das teclas possíveis é representada por números entre 0 e 0xffff. Isso inclui até mesmo caracteres em japonês, "Ç", todas as teclas do tipo Shift, Esc, Caps Lock, Ctrl e o "N" com um til do espanhol. Mas algumas coisas ficam de fora, como cirílico, símbolos árabes, vietnamitas e símbolos matemáticos especiais. Contudo, isso não será algo grave, pois podemos fornecer uma função capaz de redefinir alguns destes símbolos para valores dentro de tal intervalo. O que significa que vamos precisar também de espaço em memória para armazenar tais traduções de uma tecla para outra. Um número de 100 delas pode ser estabelecido como máximo, pois a maioria dos teclados tem menos teclas que isso.

Note que este é um problema do XLib. O SDL de qualquer forma já se atém somente à 16 bytes para representar suas teclas. Então, podemos ignorar com segurança tais traduções quando estivermos programando para a Web.

Sabendo disso, o nosso vetor de teclas e vetor de traduções pode ser declarado, bem como o vetor de teclas pressionadas. Mas além das teclas pressionadas do teclado, vamos tar o mesmo tratamento para os botões pressionados no *mouse*:

**Seção: API Weaver: Definições:**

```
  int Wkeyboard[0xffff];
#if W_TARGET == W_ELF
  static struct _k_translate{ // Traduz uma tecla em outra
    unsigned original_symbol, new_symbol;
  } _key_translate[100];
#endif
// Lista de teclas do teclado que estão sendo pressionadas e que estão
// sendo soltas:
  static unsigned _pressed_keys[20];
  static unsigned _released_keys[20];
// List de botões do mouse que estão sendo pressionados e soltos:
  static unsigned _pressed_buttons[5];
  static unsigned _released_buttons[5];
```

**Seção: Cabeçalhos Weaver (continuação):**

```
// Depois declaramos o vetor de tempo pressionado para os botões do
// mouse. Este é para o teclado:
  extern int Wkeyboard[0xffff];
#ifdef W_MULTITHREAD
  pthread_mutex_t _input_mutex;
#endif
```

A inicialização de tais valores consiste em deixar todos contendo zero como valor:

**Seção: API Weaver: Inicialização (continuação):**

```
{
```

```
  int i;
  for(i = 0; i < 0xffff; i ++)
    Wkeyboard[i] = 0;
#if W_TARGET == W_ELF
  for(i = 0; i < 100; i ++){
    _key_translate[i].original_symbol = 0;
    _key_translate[i].new_symbol = 0;
  }
#endif
  for(i = 0; i < 20; i ++){
    _pressed_keys[i] = 0;
    _released_keys[i] = 0;
  }
#ifdef W_MULTITHREAD
  if(pthread_mutex_init(&_input_mutex, NULL) != 0){ // Inicializa mutex
    perror(NULL);
    exit(1);
  }
#endif
}
```

Assim como inicializamos valores, ao término do programa, podemos precisar finalizá-los:

**Seção: API Weaver: Finalização (continuação):**

```
#ifdef W_MULTITHREAD
  if(pthread_mutex_destroy(&_input_mutex) != 0){ // Finaliza mutex
    perror(NULL);
    exit(1);
}
#endif
```

Inicializar a lista de teclas pressionadas com zero funciona porque nem o SDL e nem o XLib associa qualquer tecla ao número zero. Então podemos usá-lo para representar a ausência de qualquer tecla sendo. De fato, o XLib ignora os primeiros 31 valores e o SDL ignora os primeiros 7. Desta forma, podemos usar tais espaços com segurança para representar conjuntos de teclas ao invés de uma tecla individual. Por exemplo, podemos associar a posição 6 como sendo o de todas as teclas. Qualquer tecla pressionada faz com que ativemos o seu valor. Outra posição pode ser associada ao Shift, que faria com que fosse ativada toda vez que o Shift esquerdo ou direito fosse pressionado. O mesmo para o Ctrl e Alt. Já o valor zero deve continuar sem uso para que possamos reservá-lo para valores inicializados, mas vazios ou indefinidos. Os demais valores nos indicam que uma tecla específica está sendo pressionada.

**Seção: Cabeçalhos Weaver:**

```
#define W_SHIFT 2 // Shift esquerdo ou direito
#define W_CTRL  3 // Ctrl esquerdo ou direito
#define W_ALT   4 // Alt esquerdo ou direito
#define W_ANY   6 // Qualquer botão
```

A função que nos permite traduzir uma tecla para outra consiste em percorrer o vetor de traduções e verificar se temos uma tradução registrada para o símbolo que estamos procurando:

**Seção: API Weaver: Definições (continuação):**

```
#if W_TARGET == W_ELF
static unsigned _translate_key(unsigned symbol){
  int i;
```

```
  for(i = 0; i < 100; i ++){
    if(_key_translate[i].original_symbol == 0)
      return symbol % 0xffff; // Sem mais traduções. Nada encontrado.
    if(_key_translate[i].original_symbol == symbol)
      return _key_translate[i].new_symbol % 0xffff; // Retorna tradução
  }
  return symbol % 0xffff; // Vetor percorrido e nenhuma tradução encontrada
}
#endif
```

Agora respectivamente a tarefa de adicionar uma nova tradução de tecla e a tarefa de limpar todas as traduções existentes. O que pode dar errado aí é que pode não haver espaço para novas traduções quando vamos adicionar mais uma. Neste caso, a função sinaliza isso retornando 0 ao invés de 1.

**Seção: Cabeçalhos Weaver:**

```
int Wkey_translate(unsigned old_value, unsigned new_value);
void Werase_key_translations(void);
```

**Seção: API Weaver: Definições:**

```
int Wkey_translate(unsigned old_value, unsigned new_value){
#if W_TARGET == W_ELF
  int i;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&_input_mutex);
#endif
  for(i = 0; i < 100; i ++){
    if(_key_translate[i].original_symbol == 0 ||
       _key_translate[i].original_symbol == old_value){
      _key_translate[i].original_symbol = old_value;
      _key_translate[i].new_symbol = new_value;
#ifdef W_MULTITHREAD
      pthread_mutex_unlock(&_input_mutex);
#endif
      return 1;
    }
  }
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_input_mutex);
#endif
#else
  // Isso previne aviso de que os argumentos da função não foram
  // usados se W_TARGET != W_ELF:
  old_value = new_value;
  new_value = old_value;
#endif
  return 0;
}

void Werase_key_translations(void){
#if W_TARGET == W_ELF
  int i;
```

```
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&_input_mutex);
#endif
  for(i = 0; i < 100; i ++){
    _key_translate[i].original_symbol = 0;
    _key_translate[i].new_symbol = 0;
  }
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_input_mutex);
#endif
#endif
}
```

Uma vez que tenhamos preparado as traduções, podemos enfim ir até o loop principal e acompanhar o surgimento de eventos para saber quando o usuário pressiona ou solta uma tecla. No caso de estarmos usando XLib e uma tecla é pressionada, o código abaixo é executado. A coisa mais críptica abaixo é o suo da função `XkbKeycodeToKeysym`. Mas basicamente o que esta função faz é traduzir o valor da variável `event.xkey.keycode` de uma representação inicial, que representa a posição da tecla em um teclado para o símbolo específico associado àquela tecla, algo que muda em diferentes teclados.

**Seção: API Weaver: Trata Evento Xlib:**

```
if(event.type == KeyPress){
  unsigned int code =  _translate_key(XkbKeycodeToKeysym(_dpy,
                                                         event.xkey.keycode, 0,
                                                         0));
  int i;
  // Adiciona na lista de teclas pressionadas:
  for(i = 0; i < 20; i ++){
    if(_pressed_keys[i] == 0 || _pressed_keys[i] == code){
      _pressed_keys[i] = code;
      break;
    }
  }
  // Apesar de estarmos aqui, pode ser que esta tecla já estava sendo
  // pressionada antes. O evento de 'tecla pressionada' pode ser
  // gerado mais de uma vez se uma tecla for segurada por muito
  // tempo. Mas só podemos marcar a quantidade de tempo que ela é
  // pressionada como 1 se ela realmente começou a ser pressionada
  // agora. E caso contrário, também verificamos se o valor é
  // negativo. Se for, isso significa que a tecla foi solta e
  // pressionada ao mesmo tempo no mesmo frame. Esta seqüência de
  // eventos também pode ser gerada incorretamente se a tecla for
  // pressionada por muito tempo e precisamos corrigir se isso
  // ocorrer.
  if(Wkeyboard[code] == 0)
    Wkeyboard[code] = 1;
  else if(Wkeyboard[code] < 0)
    Wkeyboard[code] *= -1;
  continue;
}
```

Já se uma tecla é solta, precisamos removê-la da lista de teclas pressionadas e adicioná-la na lista de teclas que acabaram de ser soltas:

**Seção: API Weaver: Trata Evento Xlib:**

```
if(event.type == KeyRelease){
  unsigned int code =  _translate_key(XkbKeycodeToKeysym(_dpy,
                                                         event.xkey.keycode,
                                                         0, 0));
  int i;
  // Remove da lista de teclas pressionadas
  for(i = 0; i < 20; i ++){
    if(_pressed_keys[i] == code){
      _pressed_keys[i] = 0;
      break;
    }
  }
  for(; i < 19; i ++){// Preenche o buraco que ficou na lista após a remoção
    _pressed_keys[i] = _pressed_keys[i + 1];
  }
  _pressed_keys[19] = 0;
  // Adiciona na lista de teclas soltas:
  for(i = 0; i < 20; i ++){
    if(_released_keys[i] == 0 || _released_keys[i] == code){
      _released_keys[i] = code;
      break;
    }
  }
  // Atualiza vetor de teclado
  Wkeyboard[code] *= -1;
  continue;
}
```

O evento de pressionar uma tecla faz com que ela vá para a lista de teclas pressionadas. O evento de soltar uma tecla remove ela desta lista e faz ela ir para a lista de teclas soltas. Mas a cada *frame* temos que também limpar a lista de teclas que foram soltas no *frame* anterior. E incrementar os valores no vetor que mede o tempo em que cada tecla está sendo pressionada para cada tecla que está na lista de teclas pressionadas. Isso precisa ser feito imediatamente antes de lermos os eventos pendentes. Somente imediatamente antes de obtermos os eventos deste *frame* devemos terminar de processar todas as ocorrências do *frame* anterior. Caso contrário, ocorreriam valores errôneos e nunca conseguiríamos manter em 1 o valor de tempo para uma tecla que acabou de ser pressionada neste *frame*.

**Seção: API Weaver: Imediatamente antes de tratar eventos:**

```
{
  int i, key;
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&_input_mutex);
#endif
  // Limpar o vetor de teclas soltas e zerar seus valores no vetor de teclado:
  for(i = 0; i < 20; i ++){
    key = _released_keys[i];
    // Se a tecla está com um valor positivo, isso significa que os
    //  eventos de soltar a tecla e apertar ela de novo foram gerados
```

```
    //  juntos. Isso geralmente acontece quando um usuário pressiona uma
    //  tecla por muito tempo. Depois de algum tempo, o servidor passa a
    //  interpretar isso como se o usuário estivesse apertando e
    //  soltando a tecla sem parar. Isso é útil em editores de texto
    //  quando você segura uma tecla e a letra que ela representa começa
    //  a ser inserida sem parar após um tempo. Mas aqui isso deixa o
    //  ato de medir o tempo cheio de detalhes incômodos. Temos que
    //  remover da lista de teclas soltas esta tecla, que provavelmente
    //  não foi solta de verdade:
    while(Wkeyboard[key] > 0){
      int j;
      for(j = i; j < 19; j ++){
        _released_keys[j] = _released_keys[j+1];
      }
      _released_keys[19] = 0;
      key = _released_keys[i];
    }
    if(key == 0) break; // Chegamos ao fim da lista de teclas pressionadas
    // Tratando casos especiais de valores que representam mais de uma tecla:
    if(key == W_LEFT_CTRL || key == W_RIGHT_CTRL) Wkeyboard[W_CTRL] = 0;
    else if(key == W_LEFT_SHIFT || key == W_RIGHT_SHIFT) Wkeyboard[W_SHIFT] = 0;
    else if(key == W_LEFT_ALT || key == W_RIGHT_ALT) Wkeyboard[W_ALT] = 0;
    // Como foi solta no frame anterior, o tempo que ela está pressionada é 0:
    Wkeyboard[key] = 0;
    _released_keys[i] = 0; // Tecla removida da lista de teclas soltas
  }
  // Para teclas pressionadas, incrementar o seu contador de tempo:
  for(i = 0; i < 20; i ++){
    key = _pressed_keys[i];
    if(key == 0) break; // Fim da lista, encerrar
    // Casos especiais:
    if(key == W_LEFT_CTRL || key == W_RIGHT_CTRL)
      Wkeyboard[W_CTRL] += _elapsed_milisseconds;
    else if(key == W_LEFT_SHIFT || key == W_RIGHT_SHIFT)
      Wkeyboard[W_SHIFT] += _elapsed_milisseconds;
    else if(key == W_LEFT_ALT || key == W_RIGHT_ALT)
      Wkeyboard[W_ALT] += _elapsed_milisseconds;
    // Aumenta o contador de tempo:
    Wkeyboard[key] += _elapsed_milisseconds;
  }
}
```

Por fim, preenchemos a posição `Wkeyboard[W_ANY]` depois de tratarmos todos os eventos:

**Seção: API Weaver: Loop Principal (continuação):**

```
#ifdef W_MULTITHREAD
  pthread_mutex_lock(&_input_mutex);
#endif
Wkeyboard[W_ANY] = (_pressed_keys[0] != 0); // Se há alguma tecla pressionada
```

Isso conclui o código que precisamos para o teclado no Xlib. Mas ainda não acabou. Precisamos de macros para representar as diferentes teclas de modo que um usuário possa consultar se uma tecla está pressionada sem saber o código da tecla no Xlib:

**Seção: Cabeçalhos Weaver:**

```
#if W_TARGET == W_ELF
#define W_UP           XK_Up
#define W_RIGHT        XK_Right
#define W_DOWN         XK_Down
#define W_LEFT         XK_Left
#define W_PLUS         XK_KP_Add
#define W_MINUS        XK_KP_Subtract
#define W_ESC          XK_Escape
#define W_A            XK_a
#define W_S            XK_s
#define W_D            XK_d
#define W_W            XK_w
#define W_ENTER        XK_Return
#define W_LEFT_CTRL    XK_Control_L
#define W_RIGHT_CTRL   XK_Control_R
#define W_F1           XK_F1
#define W_F2           XK_F2
#define W_F3           XK_F3
#define W_F4           XK_F4
#define W_F5           XK_F5
#define W_F6           XK_F6
#define W_F7           XK_F7
#define W_F8           XK_F8
#define W_F9           XK_F9
#define W_F10          XK_F10
#define W_F11          XK_F11
#define W_F12          XK_F12
#define W_BACKSPACE    XK_BackSpace
#define W_TAB          XK_Tab
#define W_PAUSE        XK_Pause
#define W_DELETE       XK_Delete
#define W_SCROLL_LOCK  XK_Scroll_Lock
#define W_HOME         XK_Home
#define W_PAGE_UP      XK_Page_Up
#define W_PAGE_DOWN    XK_Page_Down
#define W_END          XK_End
#define W_INSERT       XK_Insert
#define W_NUM_LOCK     XK_Num_Lock
#define W_ZERO         XK_KP_0
#define W_ONE          XK_KP_1
#define W_TWO          XK_KP_2
#define W_THREE        XK_KP_3
#define W_FOUR         XK_KP_4
#define W_FIVE         XK_KP_5
#define W_SIX          XK_KP_6
#define W_SEVEN        XK_KP_7
```

```
#define W_EIGHT       XK_KP_8
#define W_NINE        XK_KP_9
#define W_LEFT_SHIFT  XK_Shift_L
#define W_RIGHT_SHIFT XK_Shift_R
#define W_CAPS_LOCK   XK_Caps_Lock
#define W_LEFT_ALT    XK_Alt_L
#define W_RIGHT_ALT   XK_Alt_R
#define W_Q           XK_q
#define W_E           XK_e
#define W_R           XK_r
#define W_T           XK_t
#define W_Y           XK_y
#define W_U           XK_u
#define W_I           XK_i
#define W_O           XK_o
#define W_P           XK_p
#define W_F           XK_f
#define W_G           XK_g
#define W_H           XK_h
#define W_J           XK_j
#define W_K           XK_k
#define W_L           XK_l
#define W_Z           XK_z
#define W_X           XK_x
#define W_C           XK_c
#define W_V           XK_v
#define W_B           XK_b
#define W_N           XK_n
#define W_M           XK_m
#endif
```

A última coisa que resta para termos uma API funcional para lidar com teclados é uma função para limpar o vetor de teclados e a lista de teclas soltas e pressionadas. Desta forma, podemos nos livrar de teclas pendentes quando saímos de um loop principal para outro, além de termos uma forma de fazer com que o programa possa descartar teclas pressionadas em momentos dos quais não era interessante levá-las em conta.

Mas não vamos querer fazer isso só com o teclado, mas com todas as formas de entrada possíveis. Portanto, vamos deixar este trecho de código com uma marcação para inserirmos mais coisas depois:

**Seção: Cabeçalhos Weaver (continuação):**

```
void Wflush_input(void);
```

**Seção: API Weaver: Definições (continuação):**

```
void Wflush_input(void){
#ifdef W_MULTITHREAD
    pthread_mutex_lock(&_input_mutex);
#endif
  {
 // Limpa informação do teclado
    int i, key;
    for(i = 0; i < 20; i ++){
```

```
      key = _pressed_keys[i];
      _pressed_keys[i] = 0;
      Wkeyboard[key] = 0;
      key = _released_keys[i];
      _released_keys[i] = 0;
      Wkeyboard[key] = 0;
    }
  }
```

<Seção a ser Inserida: Limpar Entrada>

```
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_input_mutex);
#endif
}
```

Quase tudo o que foi definido aqui aplica-se tanto para o Xlib rodando em um programa nativo para Linux como em um programa SDL compilado para a Web. A única exceção é o tratamento de eventos, que é feita usando funções diferentes nas duas bibliotecas.

Para um programa compilado para a Web, precisamos inserir o cabeçalho SDL:

**Seção: Cabeçalhos Weaver (continuação):**

```
#if W_TARGET == W_WEB
#include <SDL/SDL.h>
#endif
```

E tratamos o evento de uma tecla ser pressionada exatamente da mesma forma, mas respeitando as diferenças das bibliotecas em como acessar cada informação:

**Seção: API Weaver: Trata Evento SDL:**

```
if(event.type == SDL_KEYDOWN){
  unsigned int code =  event.key.keysym.sym;
  int i;
  // Adiciona na lista de teclas pressionadas
  for(i = 0; i < 20; i ++){
    if(_pressed_keys[i] == 0 || _pressed_keys[i] == code){
      _pressed_keys[i] = code;
      break;
    }
  }
    // Atualiza vetor de teclado se a tecla não estava sendo
    // pressionada. Algumas vezes este evento é gerado repetidas vezes
    // quando apertamos uma tecla por muito tempo. Então só devemos
    // atribuir 1 à posição do vetor se realmente a tecla não estava
    // sendo pressionada antes.
  if(Wkeyboard[code] == 0)
    Wkeyboard[code] = 1;
  else if(Wkeyboard[code] < 0)
    Wkeyboard[code] *= -1;
  continue;
}
```

Por fim, o evento da tecla sendo solta:

**Seção: API Weaver: Trata Evento SDL:**

```
if(event.type == SDL_KEYUP){
  unsigned int code =  event.key.keysym.sym;
```

```
  int i;
  // Remove da lista de teclas pressionadas
  for(i = 0; i < 20; i ++){
    if(_pressed_keys[i] == code){
      _pressed_keys[i] = 0;
      break;
    }
  }
  for(; i < 19; i ++){
    _pressed_keys[i] = _pressed_keys[i + 1];
  }
  _pressed_keys[19] = 0;
  // Adiciona na lista de teclas soltas:
  for(i = 0; i < 20; i ++){
    if(_released_keys[i] == 0 || _released_keys[i] == code){
      _released_keys[i] = code;
      break;
    }
  }
  // Atualiza vetor de teclado
  Wkeyboard[code] *= -1;
  continue;
}
```

E por fim, a posição das teclas para quando usamos SDL no vetor de teclado será diferente e correspondente aos valores usados pelo SDL:

**Seção: Cabeçalhos Weaver:**

```
#if W_TARGET == W_WEB
#define W_UP           SDLK_UP
#define W_RIGHT        SDLK_RIGHT
#define W_DOWN         SDLK_DOWN
#define W_LEFT         SDLK_LEFT
#define W_PLUS         SDLK_PLUS
#define W_MINUS        SDLK_MINUS
#define W_ESC          SDLK_ESCAPE
#define W_A            SDLK_a
#define W_S            SDLK_s
#define W_D            SDLK_d
#define W_W            SDLK_w
#define W_ENTER        SDLK_RETURN
#define W_LEFT_CTRL    SDLK_LCTRL
#define W_RIGHT_CTRL   SDLK_RCTRL
#define W_F1           SDLK_F1
#define W_F2           SDLK_F2
#define W_F3           SDLK_F3
#define W_F4           SDLK_F4
#define W_F5           SDLK_F5
#define W_F6           SDLK_F6
#define W_F7           SDLK_F7
#define W_F8           SDLK_F8
```

```
#define W_F9           SDLK_F9
#define W_F10          SDLK_F10
#define W_F11          SDLK_F11
#define W_F12          SDLK_F12
#define W_BACKSPACE    SDLK_BACKSPACE
#define W_TAB          SDLK_TAB
#define W_PAUSE        SDLK_PAUSE
#define W_DELETE       SDLK_DELETE
#define W_SCROLL_LOCK  SDLK_SCROLLOCK
#define W_HOME         SDLK_HOME
#define W_PAGE_UP      SDLK_PAGEUP
#define W_PAGE_DOWN    SDLK_PAGEDOWN
#define W_END          SDLK_END
#define W_INSERT       SDLK_INSERT
#define W_NUM_LOCK     SDLK_NUMLOCK
#define W_ZERO         SDLK_0
#define W_ONE          SDLK_1
#define W_TWO          SDLK_2
#define W_THREE        SDLK_3
#define W_FOUR         SDLK_4
#define W_FIVE         SDLK_5
#define W_SIX          SDLK_6
#define W_SEVEN        SDLK_7
#define W_EIGHT        SDLK_8
#define W_NINE         SDLK_9
#define W_LEFT_SHIFT   SDLK_LSHIFT
#define W_RIGHT_SHIFT  SDLK_RSHIFT
#define W_CAPS_LOCK    SDLK_CAPSLOCK
#define W_LEFT_ALT     SDLK_LALT
#define W_RIGHT_ALT    SDLK_RALT
#define W_Q            SDLK_q
#define W_E            SDLK_e
#define W_R            SDLK_r
#define W_T            SDLK_t
#define W_Y            SDLK_y
#define W_U            SDLK_u
#define W_I            SDLK_i
#define W_O            SDLK_o
#define W_P            SDLK_p
#define W_F            SDLK_f
#define W_G            SDLK_g
#define W_H            SDLK_h
#define W_J            SDLK_j
#define W_K            SDLK_k
#define W_L            SDLK_l
#define W_Z            SDLK_z
#define W_X            SDLK_x
#define W_C            SDLK_c
#define W_V            SDLK_v
```

```
#define W_B             SDLK_b
#define W_N             SDLK_n
#define W_M             SDLK_m
#endif
```

## 4.3 - Invocando o loop principal

Um jogo pode ter vários loops principais. Um para a animação de abertura. Outro para a tela de título onde escolhe-se o modo do jogo. Um para cada fase ou cenário que pode-se visitar. Pode haver outro para cada "fase especial" ou mesmo para cada batalha em um jogo de RPG.

Em cada um dos loops principais, precisamos rodar possivelmente milhares de iterações. E em cada uma delas precisamos fazer algumas coisas em comum. Imediatamente antes do loop precisamos limpar todos os valores prévios armazenados no vetor de teclado. E depois em cada iteração precisamos rodar `Wrest` para obtermos os eventos de entrada, atualizarmos várias variáveis e poder desenhar na tela.

O problema é que este tipo de coisa depende do ambiente de execução em que estamos. Por exemplo, se estamos executando um programa Linux, o seguinte loop principal seria válido:

**Arquivo:** `/tmp/dummy.c`**:**

```
// Exemplo. Não faz parte de Weaver.
while(1){
  handle_input();
  handle_objects();
  Wrest(10);
}
```

Além disso poderíamos criar uma condição explícita para sairmos do loop e entrarmos em outra logo em seguida. Mas infelizmente se estamos executando em um navegador de Internet após termos o código compilado para Javascript, isso não é possível. Um loop infinito geraria um loop no código Javascript e isso faria com que a função Javascript nunca termine. O navegador congelaria dentro do *loop* e se ofereceria para matar o script problemático, sem poder fazer coisas como desenhar na tela. Talvez o navegador não conseguisse nem mesmo detectar teclas pressionadas pelo jogador.

Portanto, não podemos deixar que o loop principal seja um loop neste caso. Ele precisa ser uma função que executa de tempos em tempos. Infelizmente, a API Emscripten requer que tal função não retorne nada e nem receba argumentos. Sendo assim, toda informação necessária para o loop principal deve estar em variáveis globais. É algo ruim, mas podemos minimizar os danos disso usando a palavra-chave `static` para limitar o escopo de nossas variáveis em cada módulo.

O que queremos então é que um programa Weaver possa ter então a seguinte forma:

**Arquivo:** `/tmp/dummy.c`**:**

```
// Exemplo. Não faz parte do Weaver.
void main_loop(void){
  // ...
  Wrest(10);
}
int main(int argc, char **argv){
  Winit();
  // Executa |main_loop| como o loop principal
  Wloop(main_loop);
  Wrest();
}
```

A função `Wloop` então executa a função que recebe como argumento em um loop infinito. E esta função deve ser definida de modo diferente dependendo de qual é o nosso ambiente de execução. A declaração dela, de qualquer forma, será a mesma:

**Seção: Cabeçalhos Weaver (continuação):**

```
  void Wloop(void (*f)(void));
```

No caso do nosso ambiente de execução ser o de um programa Linux normal, a definição da função é:

**Seção: API Weaver: Definições (continuação):**

```
#if W_TARGET == W_ELF
void Wloop(void (*f)(void)){
  Wflush_input();
  for(;;){
    f();
  }
}
#endif
```

Já se estamos no ambiente de execução de um navegador de Internet, temos preocupações adicionais. Precisamos registrar uma função como um loop principal. Mas se já existe um loop principal anteriormente registrado, precisamos cancelar ele primeiro.

**Seção: Cabeçalhos Weaver (continuação):**

```
#if W_TARGET == W_WEB
#include <emscripten.h>
#endif
```

**Seção: API Weaver: Definições (continuação):**

```
#if W_TARGET == W_WEB
void Wloop(void (*f)(void)){
  emscripten_cancel_main_loop();
  Wflush_input();
  // O segundo argumento é o número de frames por segundo:
  emscripten_set_main_loop(f, 0, 1);
  // Nunca chegamos nesta parte. Inútil colocar qualquer coisa após
  // 'emscripten_set_main_loop'.
}
#endif
```

Tudo isso significa que um loop principal nunca chega ao fim. Podemos apenas invocar outro loop principal recursivamente dentro do atual. Não há como evitar esta limitação com a atual API Emscripten que precisa usar `emscripten_set_main_loop` para ativar o loop sem interferir na usabilidade do navegador de Internet. Isso também traz a limitação de que o *loop* principal seja uma função que não retorna nada e nem recebe argumentos.

A única possibilidade de evitar isso seria se fosse possível usar clausuras (*closures*). Neste caso, poderíamos definir `Wloop` como uma macro que expandiria para a definição de uma clausura que poderia ter acesso à todas as variáveis da função atual ao mesmo tempo em que ela poderia ser passada para a função de invocaçã do loop. O único compilador compatível com Emscripten é o Clang, que até implementa clausuras por meio de uma extensão não-portável chamada de "blocos". O problema é que um bloco não é intercambiável e nem pode ser convertido para uma função. Então não seria possível passá-lo para a atual função da API Emscripten que espera uma função. O GCC suporta clausuras na forma de funções aninhadas por meio de extensão não-portável, mas o GCC não é compatível com Emscripten. Então simplesmente não temos como evitar este efeito colateral.

# 4.4 - O Mouse

Um mouse do nosso ponto de vista é como se fosse um teclado, mas com menos teclas. O Xlib reconhece que mouses podem ter até 5 botões ( `Button1` , `Button2` , `Button3` , `Button4` e `Button5` ). O SDL, tentando manter portabilidade, em sua versão 1.2 reconhece 3 botões ( `SDL_BUTTON_LEFT` , `SDL_BUTTON_MIDDLE` , `SDL_BUTTON_RIGHT` ). Convenientemente, ambas as bibliotecas numeram cada um dos botões sequencialmente à partir do número 1. Nós iremos suportar 5 botões, mas um jogo deve assumir que apenas dois botões são realmente garantidos: o botão direito e esquerdo.

Além dos botões, um mouse possui também uma posição $(x, y)$ na janela em que o jogo está. Mas às vezes mais importante do que sabermos a posição é sabermos a sua velocidade. E caso esteja se movendo, para onde ele está indo e em qual velocidade. Ambas as informações podem ser captadas por valores $(dx, dy)$ que capturam em qual posição estará no mouse em 1 segundo se ele manter o mesmo deslocamento observado entre estre frame e o anterior. Em outras palavras, estes valores sãoo os componentes de sua velocidade em *pixels* por segundo.

Em suma, podemos representar o mouse como a seguinte estrutura:

**Seção: API Weaver: Definições (continuação):**

```
struct _mouse Wmouse;
```

**Seção: Cabeçalhos Weaver:**

```
extern struct _mouse{
  /* Posições de 1 a 5 representarão cada um dos botões e o 6 *é
     reservado para qualquer tecla.*/
  int buttons[7];
  int x, y, dx, dy;
} Wmouse;
```

E a tradução dos botões, dependendo do ambiente de execução será dada por:

**Seção: Cabeçalhos Weaver:**

```
#if W_TARGET == W_ELF
#define W_MOUSE_LEFT   Button1
#define W_MOUSE_MIDDLE Button2
#define W_MOUSE_RIGHT  Button3
#define W_MOUSE_B1     Button4
#define W_MOUSE_B2     Button5
#endif
#if W_TARGET == W_WEB
#define W_MOUSE_LEFT   SDL_BUTTON_LEFT
#define W_MOUSE_MIDDLE SDL_BUTTON_MIDDLE
#define W_MOUSE_RIGHT  SDL_BUTTON_RIGHT
#define W_MOUSE_B1     4
#define W_MOUSE_B2     5
#endif
```

Agora podemos inicializar os vetores de botões soltos e pressionados:

**Seção: API Weaver: Inicialização (continuação):**

```
{ // Inicialização das estruturas do mouse
  int i;
  for(i = 0; i < 5; i ++)
    Wmouse.buttons[i] = 0;
  for(i = 0; i < 5; i ++){
    _pressed_buttons[i] = 0;
```

```
    _released_buttons[i] = 0;
  }
}
```

Imediatamente antes de tratarmos eventos, precisamos percorrer a lista de botões pressionados para atualizar seus valores e a lista de botões recém-soltos para removê-los da lista. É essencialmente o mesmo trabalho que fazemos com o teclado.

**Seção: API Weaver: Imediatamente antes de tratar eventos:**

```
{
  int i, button;
  // Limpar o vetor de botõoes soltos e zerar seus valores no vetor de mouse:
  for(i = 0; i < 5; i ++){
    button = _released_buttons[i];
    while(Wmouse.buttons[button] > 0){
      int j;
      for(j = i; j < 4; j ++){
        _released_buttons[j] = _released_buttons[j+1];
      }
      _released_buttons[4] = 0;
      button = _released_buttons[i];
    }
    if(button == 0) break;
    Wmouse.buttons[button] = 0;
    _released_buttons[i] = 0;
  }
  // Para botões pressionados, incrementar o tempo em que estão pressionados:
  for(i = 0; i < 5; i ++){
    button = _pressed_buttons[i];
    if(button == 0) break;
    Wmouse.buttons[button] += _elapsed_milisseconds;
  }
}
```

Tendo esta estrutura pronta, iremos então tratar a chegada de eventos de botões do mouse sendo pressionados caso estejamos em um ambiente de execução baseado em Xlib:

**Seção: API Weaver: Trata Evento Xlib:**

```
if(event.type == ButtonPress){
  unsigned int code =  event.xbutton.button;
  int i;
  // Adiciona na lista de botões pressionados:
  for(i = 0; i < 5; i ++){
    if(_pressed_buttons[i] == 0 || _pressed_buttons[i] == code){
      _pressed_buttons[i] = code;
      break;
    }
  }
    // Atualiza vetor de mouse se a tecla não estava sendo
    // pressionada. Ignoramos se o evento está sendo gerado mais de uma
    // vez sem que o botão seja solto ou caso o evento seja gerado
    // imediatamente depois de um evento de soltar o mesmo botão:
  if(Wmouse.buttons[code] == 0)
```

```
    Wmouse.buttons[code] = 1;
  else if(Wmouse.buttons[code] < 0)
    Wmouse.buttons[code] *= -1;
  continue;
}
```

E caso um botão seja solto, também tratamos tal evento:

**Seção: API Weaver: Trata Evento Xlib:**

```
if(event.type == ButtonRelease){
  unsigned int code = event.xbutton.button;
  int i;
  // Remove da lista de botões pressionados
  for(i = 0; i < 5; i ++){
    if(_pressed_buttons[i] == code){
      _pressed_buttons[i] = 0;
      break;
    }
  }
  for(; i < 4; i ++){
    _pressed_buttons[i] = _pressed_buttons[i + 1];
  }
  _pressed_buttons[4] = 0;
  // Adiciona na lista de botões soltos:
  for(i = 0; i < 5; i ++){
    if(_released_buttons[i] == 0 || _released_buttons[i] == code){
      _released_buttons[i] = code;
      break;
    }
  }
  // Atualiza vetor de mouse
  Wmouse.buttons[code] *= -1;
  continue;
}
```

No ambiente de execução com SDL também precisamos checar quando um botão é pressionado:

**Seção: API Weaver: Trata Evento SDL:**

```
if(event.type == SDL_MOUSEBUTTONDOWN){
  unsigned int code =  event.button.button;
  int i;
  // Adiciona na lista de botões pressionados
  for(i = 0; i < 5; i ++){
    if(_pressed_buttons[i] == 0 || _pressed_buttons[i] == code){
      _pressed_buttons[i] = code;
      break;
    }
  }
  // Atualiza vetor de mouse se o botão já não estava sendo pressionado
  // antes.
  if(Wmouse.buttons[code] == 0)
    Wmouse.buttons[code] = 1;
  else if(Wmouse.buttons[code] < 0)
```

```
    Wmouse.buttons[code] *= -1;
  continue;
}
```

E quando um botão é solto:

**Seção: API Weaver: Trata Evento SDL:**

```
if(event.type == SDL_MOUSEBUTTONUP){
  unsigned int code =  event.button.button;
  int i;
  // Remove da lista de botões pressionados
  for(i = 0; i < 5; i ++){
    if(_pressed_buttons[i] == code){
      _pressed_buttons[i] = 0;
      break;
    }
  }
  for(; i < 4; i ++){
    _pressed_buttons[i] = _pressed_buttons[i + 1];
  }
  _pressed_buttons[4] = 0;
  // Adiciona na lista de botões soltos:
  for(i = 0; i < 5; i ++){
    if(_released_buttons[i] == 0 || _released_buttons[i] == code){
      _released_buttons[i] = code;
      break;
    }
  }
  // Atualiza vetor de teclado
  Wmouse.buttons[code] *= -1;
  continue;
}
```

E finalmente, o caso especial para verificar se qualquer botão foi pressionado:

**Seção: API Weaver: Loop Principal (continuação):**

```
Wmouse.buttons[W_ANY] = (_pressed_buttons[0] != 0);
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_input_mutex);
#endif
```

## 4.4.1- Obtendo o movimento

Agora iremos calcular o movimento do mouse. Primeiramente, no início do programa devemos zerar tais valores para evitarmos valores absurdos na primeira iteração:

**Seção: API Weaver: Inicialização (continuação):**

```
  Wmouse.x = Wmouse.y = Wmouse.dx = Wmouse.dy = 0;
```

É importante que no início de cada iteração, antes de tratarmos os eventos, nós zeremos os valores $(dx, dy)$ do mouse. Caso o mouse não receba nenhum evento de movimento, tais valores estarão corretos. Já se ele receber, aí de qualquer forma teremos a chance de atualizar os valores no tratamento do evento:

**Seção: API Weaver: Imediatamente antes de tratar eventos (continuação):**

```
  Wmouse.dx = Wmouse.dy = 0;
```

Em seguida, cuidamos do caso no qual temos um evento Xlib de movimento do mouse:

**Seção: API Weaver: Trata Evento Xlib (continuação):**

```
if(event.type == MotionNotify){
  int x, y, dx, dy;
  x = event.xmotion.x;
  y = event.xmotion.y;
  dx = x - Wmouse.x;
  dy = y - Wmouse.y;
  Wmouse.dx = ((float) dx / _elapsed_milisseconds) * 1000;
  Wmouse.dy = ((float) dy / _elapsed_milisseconds) * 1000;
  Wmouse.x = x;
  Wmouse.y = y;
  continue;
}
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_input_mutex); // Último evento Xlib, libera mutex
#endif
```

Agora é só usarmos a mesma lógica para tratarmos o evento SDL:

**Seção: API Weaver: Trata Evento SDL (continuação):**

```
if(event.type == SDL_MOUSEMOTION){
  int x, y, dx, dy;
  x = event.motion.x;
  y = event.motion.y;
  dx = x - Wmouse.x;
  dy = y - Wmouse.y;
  Wmouse.dx = ((float) dx / _elapsed_milisseconds) * 1000;
  Wmouse.dy = ((float) dy / _elapsed_milisseconds) * 1000;
  Wmouse.x = x;
  Wmouse.y = y;
  continue;
}
#ifdef W_MULTITHREAD
  pthread_mutex_unlock(&_input_mutex); // Último evento SDL, libera mutex
#endif
```

E a última coisa que precisamos fazer é zerar e limpar todos os vetores de botões e variáveis de movimento toda vez que for requisitado limpar todos os buffers de entrada. Como ocorre antes de entrarmos em um loop principal:

**Seção: Limpar Entrada (continuação):**

```
{
  int i;
  for(i = 0; i < 5; i ++){
    _released_buttons[i] = 0;
    _pressed_buttons[i] = 0;
  }
  for(i = 0; i < 7; i ++)
    Wmouse.buttons[i] = 0;
  Wmouse.dx = 0;
```

```
  Wmouse.dy = 0;
}
```

# Capítulo 5: Plugins

Um projeto Weaver pode suportar *plugins*. Mas o que isso significa depende se o projeto está sendo compilado para ser um executável ELF ou uma página web.

Do ponto de vista de um usuário, o que chamamos de *plugin* deve ser um único arquivo com código C (digamos que seja `myplugin.c`). Este arquivo pode ser copiado e colado para o diretório `plugins` de um Projeto Weaver e então subitamente o projeto passa a ter acesso à duas funções. Uma delas que ativa o *plugin* (que no caso será `Wenable_plugin("myplugin")` ) e uma que desativa (no caso, `Wdisable_plugin("myplugin")` ).

Quando um *plugin* está ativo, ele pode passar a executar alguma atividade durante todo *loop* principal e também pode executar atividades de inicialização no momento em que é ativado. No momento em que é desativado, ele executa suas atividades de finalização. Um plugin também pode se auto-ativar automaticamente durante a inicialização dependendo de sua natureza.

Uma atividade típica que podem ser implementadas via *plugin* é um esquema de tradução de teclas do teclado para que teclados com símbolos exóticos sejam suportados. Ele só precisaria definir o esquema de tradução na inicialização e nada precisaria ser feito em cada iteração do *loop* principal. Ou pode ser feito um *plugin* que não faça nada em sua inicialização, mas em todo *loop* principal mostre no canto da tela um indicador de quantos *frames* por segundo estão sendo executados.

Mas as possibilidades não param nisso. Uma pessoa pode projetar um jogo de modo que a maior parte das entidades que existam nele sejam na verdade *plugins*. Desta forma, um jogador poderia personalizar sua instalação do jogo removendo elementos não-desejados do jogo ou adicionando outros meramente copiando arquivos.

Neste ponto, esbarramos em algumas limitações do ambiente Web. Programas compilados por meio de Emscripten só podem ter os tais "*plugins*" definidos durante a compilação. Para eles, o código do *plugin* deve ser injetado em seu proprio código durante a compilação. De fato, pode-se questionar se podemos realmente chamar tais coisas de *plugins*.

Já programas compilados para executáveis Linux possuem muito mais liberdade. Eles podem checar por seus *plugins* em tempo de execução, não de compilação. De fato, isso nos permite recompilar os plugins enquanto o nosso programa está executando e assim modificar um jogo em desenvolvimento sem a necessidade de interromper sua execução. Essa técnica é chamada de **programação interativa**.

## 5.1 - Interface dos Plugins

Todo *plugin*, cujo nome é `MYPLUGIN` , e cujo código está em `plugins/MYPLUGIN.c`, deve definir as seguintes funções:

- `void _init_plugin_MYPLUGIN(void)` : Esta função será executada toda vez que o *plugin* for ativado por meio de `Wenable_plugin` .
- `void _fini_plugin_MYPLUGIN(void)` : Esta função será executada toda vez que o *plugin* for desativado por meio de `Wdisable_plugin` .
- `void _run_plugin_MYPLUGIN(void)` : Esta função será executada toda vez que um *plugin* estiver ativado e estivermos em uma iteração do *loop* principal.

Pode-se usar variáveis globais estáticas, mas deve-se ter em mente que o seu conteúdo será perdido e reinicializado toda vez que se desativa e ativa o *plugin*. Como a nossa interface atualmente não suporta em *plugins* qualquer tipo de variável global ou funções com retorno, embora o *plugin* possa ler informações e variáveis do programa, ele não tem como passar qualquer tipo de informação para o programa principal. Isso talvez mude futuramente, mas o formato em que é feita uma comunicação entre o programa e *plugins* precisa ser pensado com cuidado antes de implementado.

Deve-se também tomar cuidado com funções de bibliotecas externas usadas em *plugins*. Algumas funções podem usar variáveis globais que teriam seu valor perdido ao desativar um *plugin* e ativá-lo novamente. Este também é um dos motivos pelo qual a *engine* Weaver define a sua própria versão de muitas bibliotecas.