The Weaver API

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article describes using literary programming the Weaver API. Weaver is a game engine and this API are how programmers interact with the engine in their game projects. Besides the API, this article also covers how the configuration file is interpreted and how game loops should be managed in a game project. The API is portable code which should work under OpenBSD, Linux, Windows and Web Assembly environments.*

1. Introduction

1.1. File Organization

When a user types `weaver PROJECT` at command line, a directory with a new Weaver project is created. Inside the directory, the file with the main loop is in `src/game.c` and inside them we find:

```
#include "game.h"

void main_loop(void){ // The game main loop
 LOOP_INIT: // Initialization code

 LOOP_BODY: // Code executed each iteration
    if(W.keyboard[W_ANY])
        Wexit_loop();
 LOOP_END: // Code executed at finalization
    return;
}

int main(void){
  Winit(); // Initializes Weaver
  Wloop(main_loop); // Enter a new game loop
  return 0;
}
```

And inside `src/game.h`, we find:

```
#ifndef _game_h_
#define _game_h_
```

```
#include "weaver/weaver.h"
#include "includes.h"

struct _game_struct{
  // You can customise this structure declaring variables here.
  // But don't change it's name. And access it by its pointer: W.game
  int whatever; // <- This variable is here to prevent compiler errors
} _game;

void main_loop(void);

#endif
```

In this file there's an struct which can be customized by the user and which is where variables with global states should be declared. The variables declared here are also variables which will be saved in a player's save file and which will be sent over a network to inform clients about the game state. This struct should be referenced by the variable `W.game`. This also gives us information that the API will be organized in a way that will exist a `struct` called `W` where the API resources will be centralized.

The file `includes.h` is just a header which includes in the project all the other headers of modules created by the user (each module is a C source file and a header).

All the API code shall be present or be included by macros in the files `weaver.c` and `weaver.h`. The `weaver.h` organization is:

File: `project/src/weaver/weaver.h`:

```
#ifndef _weaver_h_
#define _weaver_h_
#ifdef __cplusplus
  extern "C" {
#endif
#include "../../conf/conf.h"
                    <Section to be inserted: Global Structure>
                    <Section to be inserted: Weaver Headers>
                    <Section to be inserted: Weaver Macros>
#ifdef __cplusplus
  }
#endif
#endif
```

We include in the header the configuration file `conf.h` responsible to control the settings of our engine.

1.2. The W Structure

The global structure referenced in the previous code is a `struct` called `W`. We already mentioned it in the code comments that the `struct _game_struct _game` definded in `game.h`. Now we can start to define this structure:

Section: Global Structure:

```
// This structure will contain all the variables and functions defined at
// Weaver API:
extern struct _weaver_struct{
```

```
  struct _game_struct *game;
                    <Section to be inserted: Weaver Variables>
                    <Section to be inserted: Weaver Functions>
} W;
```

Besides `W.game`, there will be other variables in this structure. We basically will centralize all the public functions of our API here. Only private functions whose names starts with "_" and some initialization and finalization functions won't be inside the `W` structure. In this way we prevent the pollution of the global namespace.

We will also define here the general structure of our `weaver.c` file:

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

```
#include "weaver.h"
#include "../game.h"
              <Section to be inserted: Weaver API: Internal Header>
              <Section to be inserted: Weaver API: Static Variables>
                <Section to be inserted: Weaver API: Definitions>
                <Section to be inserted: Weaver API: Functions>
                  <Section to be inserted: Weaver API: Base>
```

In "Weaver API: Definitions" we will declare new kind of structures. The first thing will be the `W` structure, which we already declared in the header:

Section: Weaver API: Definitions:

```
struct _weaver_struct W;
```

In "Weaver API: Functions" we will put almost all our functions. Except some base functions which can't or shouldn't be put inside the `W` structure, like initialization and finalization functions.

1.3. Initialization and Finalization Functions

One thing that the initialization function must do is initialize the values inside the `W` structure:

Section: Weaver Headers:

```
void Winit(void);
```

Section: Weaver API: Base:

```
void Winit(void){
  W.game = &_game;
                  <Section to be inserted: Weaver API: Initialization>
}
```

The finalization function shall deallocate any pending memory, end resource usage and close the program informing if everything happened as expected:

Section: Weaver Headers:

```
void Wexit(void);
```

Section: Weaver API: Base:

```
void Wexit(void){
                  <Section to be inserted: Weaver API: Finalization>
  exit(0);
}
```

The `exit` function requires the inclusion of the standard header:

Section: Weaver Headers:

```
#include <stdlib.h>
```

The rest of the code executed in initialization and finalization will be described along the article.

2. Time Counting

Weaver measeures elapsed time in microseconds (10^{-6}s) and stores the time counting in at least 64 bits of memory. Besides the total elapsed time since the program initialization, we also store the time difference between the current iteration in the main loop and the previous one.

First we neet a place to store our last time measure and we use a global variable. In Windows we use a specific type to store large integers (`LARGE_INTEGER`) and in other systems we use a `timeval` structure to store the time measure at high resolution.

Section: Weaver Headers:

```
#if defined(_WIN32)
#include <windows.h>
LARGE_INTEGER _last_time;
#else
#include <sys/time.h>
struct timeval _last_time;
#endif
```

The idea is store in this variable always the last time measure. It's initialized with our first time measure at initialization:

Section: Weaver API: Initialization:

```
#if defined(_WIN32)
QueryPerformanceCounter(&_last_time);
#else
gettimeofday(&_last_time, NULL);
#endif
```

After initialization, all other updates in this variable will happen using the following declared function:

Section: Weaver Headers (continuation):

```
unsigned long _update_time(void);
```

This function will read thecurrent time and store the variable. It will always return the difference in microseconds between the last two measures. In Unix systems we compute the time difference using the method recommended in the GNU C Library manual. This subtraction method is more portable and works even if the `timeval` elements are stored as "unsigned". The disadvantage is that the code is less clear and intuitive. The code is:

Section: Weaver API: Definitions:

```
#if !defined(_WIN32)
unsigned long _update_time(void){
  int nsec;
  unsigned long result;
  struct timeval _current_time;
  gettimeofday(&_current_time, NULL);
  // Performing the carry:
  if(_current_time.tv_usec < _last_time.tv_usec){
    nsec = (_last_time.tv_usec - _current_time.tv_usec) / 1000000 + 1;
    _last_time.tv_usec -= 1000000 * nsec;
    _last_time.tv_sec += nsec;
  }
  if(_current_time.tv_usec - _last_time.tv_usec > 1000000){
```

```
    nsec = (_current_time.tv_usec - _last_time.tv_usec) / 1000000;
    _last_time.tv_usec += 1000000 * nsec;
    _last_time.tv_sec -= nsec;
  }
  if(_current_time.tv_sec < _last_time.tv_sec){
    // Overflow
    result = (_current_time.tv_sec - _last_time.tv_sec) * (-1000000);
    // This is always positive:
    result += (_current_time.tv_usec - _last_time.tv_usec);
  }
  else{
    result = (_current_time.tv_sec - _last_time.tv_sec) * 1000000;
    result += (_current_time.tv_usec - _last_time.tv_usec);
  }
  _last_time.tv_sec = _current_time.tv_sec;
  _last_time.tv_usec = _current_time.tv_usec;
  return result;
}
#endif
```

At Windows systems, there's already a function that measure time in microseconds. So the function becames much simpler:

Section: Weaver API: Definitions:

```
#if defined(_WIN32)
unsigned long _update_time(void){
  LARGE_INTEGER prev;
  prev.QuadPart = _last_time.QuadPart;
  QueryPerformanceCounter(&_last_time);
  return (_last_time.QuadPart - prev.QuadPart);
}
#endif
```

3. The main loops.

All games are organized inside main loops. They are code which iterates indefinitelly until some condition send the program for another main loop or terminate the program.

As shown in the initial code at `game.c`, a main loop should be declared as:

```
void nome_do_loop(void){
 LOOP_INIT: // Code executed at initialization

 LOOP_BODY: // Code executed in each iteration
    if(W.keyboard[W_ANY])
        Wexit_loop();
 LOOP_END: // Code executed at finalization
    return;
}
```

Before understanding how we should enter correctly in a main loop, it's important to describe how the loop is executed. Note that it has an initialization section, an execution section and a finalization section. These sections are delimited by labels in upper case.

Interpreting this is very simple. You can see the code above as:

```
void nome_do_loop(void){
```

```
  // LOOP_INIT
  for(;;){
    // LOOP_BODY
    if(W.keyboard[W_ANY])
        Wexit_loop();
  }
  // LOOP_END
}
```

While this interpretation is suitable in some contexts, this is not how the main loop code is translated. We can't run an infinite loop in all environments without blocking tha game interface. In Web Assembly environments, a game loop can be executed only if they are correctly declared as such and these functions shouldn't have an infinite loop, instead they are called successively.

Because of such differences, to create more portable code, we should interpret a main loop execution as:

```
for(;;)
  nome_do_loop();
```

Inside the main loop function, we don't put an explicit loop. Instead, we decide which section of the function we should execute with the help of the labels inserted. Such labels are in fact macros with aditional logic inside and with some `goto` to decide which section should be executed.

Because of this, we can't declare variables in a main loop initialization. If so, they would have the correct value only during the first iteration, not in the others. For example, the following code would have an undefinded result and perhaps it wouldn't print anything in the screen:

```
// WRONG
void loop(void){
LOOP_INIT:
  int var = 5;
LOOP_BODY:
  if(var == 5)
    printf("var == 5\n");
LOOP_END:
}
```

But the following code is correct and print in the screen in all iterations because the variable is declared outside the function:

```
// CORRECT
static int var;

void loop(void){
LOOP_INIT:
  var = 5;
LOOP_BODY:
  if(var == 5)
    printf("var == 5\n");
LOOP_END:
}
```

Other thing that should be considered is that in fact there's not just one main loop being executed in a given moment, but two of them. One of the loops execute in a fixed frequency: the loop handling physics and game logic. The other main loop runs as fast as it: the loop rendering graphics in the screen.

Ideally in each physics and logic loop iteration, we execute one or more iterations of rendering loop. It means that we can render with a frequency greater than we move objects, detect collisions and read user input. And in each of the rendering loop iteration, we need to render different images, otherwise there's

no point in running this loop faster than the physics loop. So in the rendering loop we also interpolate the objects positions, knowing their current speed, acceleration and position.

And ensuring that our game physics and logic runs always in a fixed interval, we ensure the necessary determinism for synchronizing games in networks like the Internet. And at same time, rendering as fast as we can with interpolation gives us a more pleasant and natural experience.

For more details of how to implement thism you can check [Fiedler 2004]. Our implementation swill be like in this reference, except that our code will be much less explicit because it will be hiden by macros without explicit loops.

Let's define in the following subsections what exactly we will put in the macros present in every main loop.

3.1. Loop Initialization.

This is what the macro `LOOP_INIT` does:

First it checks variables to determine if we should finish the loop. If so, but we still have resources being loaded (images, videos, shaders, sounds), we just return from the function. If we have nothing being loaded but we still didn't executed the finalization section, we use `goto` and go to the finalization section. If there's nothing being loaded and we already ran the finalization, we finally stop the loop.

If we don't need to finish the loop, but this function is being called for the first time, we just continue the execution. Otherwise, we use a `goto` to avoid executing more than once the initialization section. Finally, if we are still here, it's because we are running the function for the first time. So we make the variable `W.loop_name` represents a string with the name of current main loop.

How do we know if we should keep executing the loop? We use a global variable. As there's only one main loop, we don't need to protect it with semaphores. The same can be done to know if we are executing a main loop for the first time, if we are in the beginning of a loop of if we already executed the finalization. Let's declare the variables:

Section: Weaver Headers (continuation):

```
#include <stdbool.h>
bool _running_loop, _loop_begin, _loop_finalized;
```

And let's initialize them:

Section: Weaver API: Initialization (continuation):

```
_running_loop = false;
_loop_begin = false;
_loop_finalized = false;
```

Know if we are still loading resources (usually reading files) or the name of the current loop is useful not only for the engine's internal logic, but also for the user. Knowing if we are still loading files permits showing a loading screen. Knowing the current loop name is useful for debugging and for loading different resources depending of the loop. Because of this, both variables shall be declared in `W` structure. The maximum loop name which we can store can be setted with the macro `W_MAX_LOOP_NAME`.

Section: Weaver Variables (continuation):

```
// Inside W structure:
#if !defined(W_MAX_LOOP_NAME)
#define W_MAX_LOOP_NAME 64
#endif
unsigned pending_files;
char loop_name[W_MAX_LOOP_NAME];
```

During initialization we set these variables as 0 and `NULL` respectivelly:

Section: Weaver API: Initialization (continuation):

```
W.pending_files = 0;
```

```
W.loop_name[0] = '\0';
```

The function which exits the loop is:

Section: Weaver Headers (continuation):

```
#if !defined(_MSC_VER)
void _exit_loop(void) __attribute__ ((noreturn));
#else
__declspec(noreturn) void _exit_loop(void);
#endif
```

But we still won't define it. By the header, this is a function which never returns, specified as such in the above code using notation from GCC and Clang and from Visual Studio. It's so because when it exists, it just calls previous main loop which never returns, or it just exists the program, depending of the case.

After desscribing thism we can finally define the loop initialization macro:

Section: Weaver Headers (continuation):

```
#define LOOP_INIT                                                    \
  if(!_running_loop){                                                \
    if(W.pending_files)                                              \
      return;                                                        \
    if(!_loop_finalized){                                            \
      _loop_finalized = true;                                        \
      goto _LOOP_FINALIZATION;                                       \
    }                                                               \
    _exit_loop();                                                    \
  }                                                                  \
  if(!_loop_begin)                                                    \
    goto _END_LOOP_INITIALIZATION;                                    \
  snprintf(W.loop_name, W_MAX_LOOP_NAME, "%s", __func__);            \
  _BEGIN_LOOP_INITIALIZATION
```

We end with th identifier `_BEGIN_LOOP_INITIALIZATION`, which is the true label name from this macro (remember, this macro is always invoked as "`LOOP_INIT:`" with ":") .

3.2. Loop Body.

This is what the macro `LOOP_BODY` does:

When we are in this macro, we shall set as false the information that this is our first loop execution avoiding to re-execute the initialization again. Then, we put an `goto` after a `if` which never will be executed. This is done just so we can use the label `_BEGIN_LOOP_INITIALIZATION` with no compiler warnings of unused label. We mark with the label `_END_LOOP_INITIALIZATION` the real beginning of our loop body. We then measure the elapsed time since the last loop and store in an accumulator. If this accumulator holds a value bigger than the time expected between execution of physics code and logic code, we execute thee code. Otherwise, we just ignore these code and jump for finalization where we just render in the screen. If a lot of time passed since last loop execution, we could run more than once this loop body code.

The accumulator which determines if we should run physics code or logic code will be called `_lag`. It is a global variable:

Section: Weaver Headers (continuation):

```
unsigned long _lag;
```

This is ts initialization:

Section: Weaver API: Initialization (continuation):

```
_lag = 0;
```

We need some variables which could be read by an user with information about time. One of them (`W.t`) will store the elapsed microseconds since the game initialization. Another one (`W.dt`) will hold the interval between executions of our physics engine and game logic. Both variables needs to be declared inside `W` struct:

Section: Weaver Variables (continuation):

```
unsigned long long t;
unsigned long dt;
```

The first variable obviously shall be initialized as zero. The second one shall be initialized with the same value than macro `W_TIMESTEP`, which could be defined by the user to cntrol the granularity of code execution in physics and logic. If this macro is not defined, we assume 40000 microsecons. This ensures a frequency of 25 Hz for the execution of physics engine.

Section: Weaver API: Initialization (continuation):

```
#if !defined(W_TIMESTEP)
#define W_TIMESTEP 40000
#endif
W.dt = W_TIMESTEP;
W.t = 0;
```

The code of physics engine and internal logic shall be encapsuled in a function called `_update`:

Section: Weaver Headers (continuation):

```
void _update(void);
```

We won't at this moment define the code in this function:

Section: Weaver API: Base (continuation):

```
void _update(void){
                <Section to be inserted: Code to execute every loop>
}
```

With all these definitions, we already can define our macro which marks the beginning of the main loop:

Section: Weaver Headers (continuation):

```
#define LOOP_BODY                                          \
  _loop_begin =  false;                                    \
  if(_loop_begin)                                          \
    goto _BEGIN_LOOP_INITIALIZATION;                       \
_END_LOOP_INITIALIZATION:                                  \
  _lag += _update_time();                                  \
  while(_lag >= W.dt){                                     \
    _update();                                             \
_LABEL_0
```

Notice that the previous macro opens a `while` loop, but don't close it. It should be closed by the code inserter by our macro which delimits the end of loop body. This macro also should decrement the variable `_lag` to prevent an infinite loop.

3.3. Loop Finalization.

This is what our macro `LOOP_END` will do:

First to ensure that the loop from last macro ends, we decrement from `_lag` the value of `W.dt`. Next we increment `W.t` with that ammount of microseconds. And next we end the

block of the loop. Outside the loop we keep doing activities not related with the physics engine and game logic. As these activities are tipically about rendering, we put them in a function called `_render`. Next we return. After the return we put a `goto` to a label which never will be executed to protect us from compiler warnings. Finally, we put a label signaling the beginning of finalization.

The only new thing here is the rendering function:

Section: Weaver Headers (continuation):

```
void _render(void);
```

For now we are not defining the code inside this function:

Section: Weaver API: Base (continuation):

```
void _render(void){
                    <Section to be inserted: Render Code>
}
```

And now we define the macro code:

Section: Weaver Headers (continuation):

```
#define LOOP_END                                    \
    _lag -=  40000;                                 \
    W.t +=  40000;                                  \
  }                                                 \
  _render();                                        \
  return;                                           \
  goto _LABEL_0;                                    \
_LOOP_FINALIZATION
```

3.4. Entering in a main loop.

Frequently we will change which main loop we will execute during the game execution. But not always this means a complete substitution. Some main loops runs inside other main loops. For example, when we enter a configuration menu during the game. Or when in a classic turn-based japanese RPG we enter in the combat loop after a random encounter.

We can use a stack of main loops, where when we exit the last main loop, we enter the next loop in the stack.

So exists two forms of entering a main loop. In one of them, using a function which we will call `Wloop` and the other with the second function `Wsubloop`. In the first we substitute one loop for another, and we can doo this only if there are no pending files being loaded (in this case, usually we will be in a loading screen, but not necessarily). We can ensure this using tricks with macros:

Section: Weaver Headers (continuation):

```
#if !defined(_MSC_VER)
void _Wloop(void (*f)(void)) __attribute__ ((noreturn));
void Wsubloop(void (*f)(void)) __attribute__ ((noreturn));
#else
__declspec(noreturn) void _Wloop(void (*f)(void));
__declspec(noreturn) void Wsubloop(void (*f)(void));
#endif
#define Wloop(a) ((W.pending_files)?(false):(_Wloop(a)))
```

The two functions to enter main loops never will return. We specified this above using both Clang and GCC conventions and also Visual Studio convention.

We also need a stack of main loops, which will be stored as an array of pointers for functions. The array will have `W_MAX_SUBLOOP` positions.This macro will be controlled by the user, but if it is not defined, we will assume 3. The choice is completely arbitrary. Besides the stack, we need a variable to store the depth of main loop stack: (`_number_of_loops`).

We declare these variables as:

Section: Weaver Headers (continuation):

```
#if !defined(W_MAX_SUBLOOP)
#define W_MAX_SUBLOOP 3
#endif
int _number_of_loops;
void (*_loop_stack[W_MAX_SUBLOOP]) (void);
```

And initialize the counting of number of loops as zero:

Section: Weaver API: Initialization (continuation):

```
_number_of_loops = 0;
```

Entering in a main loop using `Wloop` means verifying if we are already in a main loop or not. If we are, we cancel the previous loop. Next, we load he next loop to the stack and adjust the counting of main loops. We also update our time measuring and finally run the loop. In Web Assembly running in web browsers, this means calling directly a function which determines the program main loop. In other environments, we just run the function in a um `while`:

Section: Weaver API: Base (continuation):

```
void _Wloop(void (*f)(void)){
  if(_number_of_loops > 0){
                    <Section to be inserted: Cancel Main Loop>
    _number_of_loops --;
  }
              <Section to be inserted: Code Before Loop and Subloop>
              <Section to be inserted: Code before Loop, not Subloop>
  _loop_stack[_number_of_loops] = f;
  _number_of_loops ++;
#if defined(__EMSCRIPTEN__)
  emscripten_set_main_loop(f, 0, 1);
#else
  while(1)
    f();
#endif
}
```

Cancelling a main loop involves calling a function if we are running in Web Assembly environment:

Section: Cancel Main Loop:

```
#if defined(__EMSCRIPTEN__)
emscripten_cancel_main_loop();
#endif
```

Start a new subloop is similar. But we update differently our main loop counter, as it needs to be incremented. And we also need to check for stack overflows:

Section: Weaver API: Definitions (continuation):

```
void Wsubloop(void (*f)(void)){
#if defined(__EMSCRIPTEN__)
```

```
    emscripten_cancel_main_loop();
#endif
```

<Section to be inserted: Code Before Loop and Subloop>
<Section to be inserted: Code before Subloop, not Loop>

```
  if(_number_of_loops >= W_MAX_SUBLOOP){
    fprintf(stderr, "Error: Max number of subloops achieved.\n");
    fprintf(stderr, "Please, increase W_MAX_SUBLOOP in conf/conf.h"
            " to a value bigger than %d.\n", W_MAX_SUBLOOP);
    exit(1);
  }
  _loop_stack[_number_of_loops] = f;
  _number_of_loops ++;
#if defined(__EMSCRIPTEN__)
  emscripten_set_main_loop(f, 0, 1);
#else
  while(1)
    f();
#endif
}
```

Let's include a header for error message printing:

Section: Weaver Headers (continuation):

```
#include <stdio.h>
```

One necessary update before entering a main loop updating `_running_loop`, variable which tells the game engine that we should run the current game loop, not exit it. We also update a variable which stores if we are in the beginning of loop, and so needs to run initialization code. And also the variable which stores that we never executed this loop finalization. These are variables which control the main loop flow. We also update our time measure.

Section: Code Before Loop and Subloop:

```
_running_loop = true;
_loop_begin = true;
_loop_finalized = false;
_update_time();
```

3.4. Exit from Main Loop.

As there are two functions to enter in a main loop, there are two functions to exit. One of them exits the current main loop returning to the previous in the stack if it exists. The other exits all the main loops and stops the program.

The function to exit from all main loops was already partially defined and is the `Wexit` function.

Exit from a main loop involves adjust the global variable which stores if we should execute the current main loop or exits from it:

Section: Weaver Headers (continuation):

```
#define Wexit_loop() (_running_loop = false)
```

If you check again the code inserted by macros in main loops, you will notice that if this variable is false, if there are no pending files being loaded, then the function `_exit_loop` is called:

Section: Weaver Headers (continuation):

```
#if !defined(_MSC_VER)
```

```
void _exit_loop(void) __attribute__ ((noreturn));
#else
__declspec(noreturn) void _exit_loop(void);
#endif
```

The code of this function involves cancelling the current main loop and checking if there is another main loop in the stack. If not, the program exits. If exists, we run again the initialization code in that main loop and start running it:

Section: Weaver API: Definitions (continuation):

```
void _exit_loop(void){
  if(_number_of_loops <= 1){
    Wexit();
    exit(1); // This line prevents compiler warnings
  }
  else{
                  <Section to be inserted: Code after Exiting Subloop>
    _number_of_loops --;
                <Section to be inserted: Code Before Loop and Subloop>
#if defined(__EMSCRIPTEN__)
    emscripten_cancel_main_loop();
    emscripten_set_main_loop(_loop_stack[_number_of_loops - 1], 0, 1);
#else
    while(1)
      _loop_stack[_number_of_loops - 1]();
#endif
  }
}
```

Memory Management

The memory management will use our subsystem with its own allocation and management functions. One difference between our subsystem and the traditional memory management with functions `malloc` and `free` is that here we need to inform the maximum ammount of memory that we will need.

The maximum numbe of memory used will change according with the project. A game mede to a video-game console could just use the maximum quantity of available memory. A project made to run in a PC could begin in the development with a small quantity and this quantity could be doubled each time th project grows and require more memory, until achieving a maximum quantity, where the developers will try to decrease the memory consumption. The maximum memory in a PC game will be the expected specifications for target consumer machines when the game is finished.

We expect that the user will inform the maximum quantity of memory using the macro `W_MAX_MEMORY` in `conf/conf.h`. If the macro is not defined, we choose the small value of 4 MiB. This will force the user to specify a more realistic value after some time, except in the most trivial projects.

Section: Weaver Headers (continuation):

```
#ifndef W_MAX_MEMORY
#define W_MAX_MEMORY 4096
#endif
```

The maximum quantity of informed memory will be allocated during initialization and we will return the address to the pointer below:

Section: Weaver API: Static Variables:

```
static void *memory_arena;
```

To use the functions in the memory subsystem, we include the header:

Section: Weaver API: Internal Header:

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

And during the initialization we allocate in our internal memory arena all the memory that we will need along the project execution:

Section: Weaver API: Initialization (continuation):

```
memory_arena = _Wcreate_arena(W_MAX_MEMORY);
```

During the finalization we free the allocated memory obtained during the initialization. But before this we free any other structure that may be allocated during the initialization. We invoke function `_Wtrasj` to do this: free everything in the right memory stack (to clean the right memory stack we pass argument 1), the place reserved to internal allocations in our API:

Section: Weaver API: Finalization:

```
_Wtrash(memory_arena, 1);
_Wdestroy_arena(memory_arena);
```

Before entering in a main loop, we use the function `_Wmempoint`. This function saves the state of our memory so that in the future we can return to this state. Doing this we can disallocate at once everything that will be allocated during the main loop.

As saving the state require knowing the memory byte alignment recommended in our environment. Weaver will get this value from macro `W_MEMORY_ALIGNMENT` that can be defined explicitly in `conf/conf.h`. If this macro is not defined, we use as default the size of a `unsigned long` as a guess.

Section: Weaver Headers (continuation):

```
#ifndef W_MEMORY_ALIGNMENT
#define W_MEMORY_ALIGNMENT (sizeof(unsigned long))
#endif
```

Knowing this we can save the state in our memory (the left and right stack):

Section: Code Before Loop and Subloop (continuation):

```
_Wmempoint(memory_arena, W_MEMORY_ALIGNMENT, 0);
_Wmempoint(memory_arena, W_MEMORY_ALIGNMENT, 1);
```

This means that when we exit from this loop, we will restore the memory state to how it was before entering in the loop. Both on left as in right stack:

Section: Cancel Main Loop (continuation):

```
_Wtrash(memory_arena, 0);
_Wtrash(memory_arena, 1);
```

This basicly willl be the garbage collector in our project. We will define now a function to the user use our memory allocation. All user allocations will be done in the left stack using the default byte alignment:

Section: Weaver API: Functions:

```
static void *_alloc(size_t size){
  return _Walloc(memory_arena, W_MEMORY_ALIGNMENT, 0, size);
}
```

We will decalre a pointer for this function in struct `W` and this function will be invoked as `W.alloc()`:

Section: Weaver Functions:

```
void *(*alloc)(size_t);
```

And this function will be ready to be used after the initialization:

Section: Weaver API: Initialization (continuation):

```
W.alloc = _alloc;
```

Empty Definitions

The following section and definitions are intentionally blank and are here to ensure correct compilation of the project. This part won't exist in the final version of this document and the currently blank parts after will be written and better explained.

Section: Code before Subloop, not Loop:

Section: Code after Exiting Subloop:

Section: Code to execute every loop:

Section: Render Code:

Section: Code before Loop, not Subloop:

Section: Weaver Macros:

References

Fiedler, G. (2004) "Fix Your Timestep!", acessado em `https://gafferongames.com/post/fix_your_timestep/` em 2020.