

# Strip Club Wars Modding Guide – version 2.0

This document started as my own documentation on how to create the various components that make up the game. I've taken out all the info about the backend C#/Unity logic and what's left is essentially a modders guide. It's also my own guide for building the run-time environment so it should be properly maintained and up to date.

**Note: This is for version 2 of the Strip Club Wars game. The TFL language has changed slightly between versions and this document reflects these changes.**

There are several sections to this document:

1. The run-time scene code, in the TFL language.
2. The character image system
3. The localization strings
4. The platform API.
5. The data files
6. Existing TFL Functions
7. Additional Information.

In addition to this document, there's a cheat sheet file that gets created by the code when it runs in dev mode. It contains a list of all the built-in functions available as well as the various key codes for the objects in the game.

## 1. Run Time Scene Code

The code for Strip Club Wars is divided into two parts. The first one is what I refer to as the platform. This is the C# code integrated into the Unity engine. This code is not available for modding in any way. The second part of this is what I refer to as the “scene” code. This is what creates the scenes that the player interacts with. It also controls most of the behavior of the NPCs, but not all. There are some notable exceptions made so for performance reasons.

The scene code is programmed using a mid-level programming language I invented years ago for a similar purpose. Which since has been modified slightly to better meet the requirements of this game. I call the language TFL, short for Total Fluke Language. Ding me some points for lack of originality. Originally my goals were less ambitious and the language was going to be much simpler which will explain some of the current limitations of the language. The reason I even started doing this was so that I could get help writing scenes from less technically savvy people and eventually if this proved to be successful, it could foster a modding community. It also kept the C# code cleaner as the TFL code could be abstracted to eliminate a lot of the error checking and verbosity that plagues higher end languages like C#.

However, as time went by I realized the additional benefits of putting code in TFL vs C#. Most importantly is the ability to change the code on the fly without having to leave a current game and go thru the recompile process. This is a huge time saver for debugging the scene code. The TFL language ecosystem then grew in complexity but it also acquired a number of idiosyncrasies that can be either minor annoyances or drive you insane. At the end of this document I'll list the most common of those gotchas so that anyone else can know what too look for. It is worth noting that some of the most annoying of these have been addressed in version 2.0.

The TFL code is compiled when the game starts into TFO files (Totla Fluke Object code). The TFO code is also ascii code, but much simplified and resembles old-time assembly languages. It's the TFL code broken up into much smaller and simpler statements so that it can be parsed and executed much faster. For the most part you don't need to look at these files. However if something is not working right, looking at it may explain why not.

## **Basic Syntax**

The TFL language is pretty standard procedural language, with Perl being probaly the most similar language. In fact, when using the Visual Studio Code editor I select Perl as the language for code highlighting and it works pretty well.

The code is structured as a collection of modules. There's one module per file, and the entire module must be defined in a single file. Each module contains a number of callables. Callables are essentially functions, but there's 4 types of them (one of which is called a function), so the term callable is used to refer to all of them.

Modules are defined with the module keyword like:

**module** ^*module\_name*

The *module\_name* is an identifier that follows the same rules as all other variable identifiers as listed in the next section.

There are three classes of variables: scalars, objects and containers. TFL is not a strongly typed language. Scalar values are automatically converted to another type when it makes sense to do so (and sometimes when it doesn't). Containers is essentially a combination of array and hash (or list and directory if using C# terminology). It's essentially the same object that can be treated as both a list and a hash. Objects are just pointers to some of the data structures used in the platform code. The TFL code can manipulate some (but not all) of data stored in these object only via built-in function calls or properties.

Comments can be included anywhere in the code. The start with the pound sign (#) and the comment lasts till the end of the line.

String literals are enclosed in double quotes. There's no escaping for newlines or double quotes.

Almost all numbers are treated as floating points and most mathematical operations will be done using floating point math.

All syntax is case-sensitive. In general everything should be in lower case except for character ids which are all uppercase.

Commands must be separated with semi-colons. Command blocks must be enclosed in curly braces. Indentation is not required, but it's always a good idea properly indent for readability. In general you don't want to break staments into multiple lines. It works in some cases but not guaranteed to work in every case.

## **Identifiers and Variables**

Identifiers include a one-character prefix and a name. With the exception of character objects and constants, all identifier names must consist of lower case letters, underscores and numbers. The first character must be a lower case letter.

Constant indentifiers have a prefix ^ and these refer to constants outside the TFL code. There's no

provision on TFL to define your own constants, either global or local. There are two types of constants: run-time constants and compile-time constants. Run-time constants are values defined in the config.dat file or the user\_config.dat file. Changing the value in the file will make the value in the code change when the config.dat file is read again (at this time there's no way to reload the file on the fly, but that could change). These are called run-time because they are evaluated at run-time (i.e. the values are not compiled into the object code).

Compile-time constants are fixed at compile time and are stored as their actual values in the object code. These refer to constants defined in the C# code. A very small number of those constants are exposed via this method. This includes things like days per week or ticks per day. The list of such constants available is in the cheat sheet.

Scalar variables are identified with a \$ prefix, so \$xyz, \$a123 and \$this\_thing are all valid scalar variable names. Scalars can be floating point numbers, strings or boolean values.

There are 7 types of objects supported at this time: character, relation, club, room, job, gossip and pregnancy. Characters have an optional @ prefix while the rest required a % prefix. So %x, %club\_x, %bar\_room and %this\_job are all objects. There's no error checking done at compile time to ensure the code is not mixing up objects of different types. But some of that will happen at run-time.

As mentioned earlier, characters have special rules. In their case the identifier name has to be all uppercase. And the prefix is optional there. So you can say @A, @BOSS or simply P are all valid character names. The @ prefix is optional but can be specified when it makes the code more readable. It is also required in some cases, like when defining a character container. Using P and then @P in the same module is allowed and both refer to the same variable.

Containers look just like other variables except they can have an index definition. So \$x[\$y] refers to the element in the \$x container with index \$y. Containers can contain objects too, so @P[] refers to a container of characters. To declare an object container, use the **cont** keyword followed by the name and the brackets, like:

```
cont @P[];
```

```
cont %r[];
```

Some built-in functions can operate on the entire container.

Note that internally I use the term “Person” instead of Character. While the API documentation will typically use character, there may be a few places that refer to persons instead. Both terms are equivalent and can be used interchangeably. (Person is used in the C# code cause Character is too long and too closely tied to “char” which is a reserved word).

A fourth type of identifier is for callables. These aren't variables, but they have the same rules for identifiers. Their names must be all lower case letters, numbers and underscores. They must begin with a lower case letter and need a & prefix. There are 4 types of callables:

1. Functions
2. Actions
3. Interactions
4. Events

While all variables are always local, callables can be accessed from other modules. One can refer to a callable in another module by including the module name. For example, &foo::bar refers to the “bar” function in the module “foo”. The module name part is not required if the callable is defined in the

same module where it's called from.

Callable names must be unique in a module, even across all types. If you have a function called `&foo`, you can't have an action with the same name. You can have a variable and function with the same name though.

Finally, identifier names can not be the same as reserved words. For example, **if** is a reserved word so you can't have a variable called `$if` or a function called `&if`.

Variables must be declared before they can be used. The declaration looks like:

```
var $var_name1, $var_name2, $var_name3;
```

```
char @X, @P;
```

```
obj %my_club;
```

```
cont $stuff[];
```

In all cases, variables of the same class may be declared on the same line, separated by commas. There are two cases where variables are declared implicitly and do not require an explicit declaration statement. We'll address those below. Note that the declaration for all non-character objects is the same.

All variables are scoped at the callable top level. Variables can not be declared inside blocks.

## ***Expressions and Assignment***

Expressions are one of the areas that may cause the most confusion in TFL. That's because there's no operator precedence. Expressions are evaluated left to right. With version 2.0, parentheses are allowed to give some precedence to some sub-expressions. But outside of that, they are all evaluated left to right.

The standard binary numeric operations are supported: `+`, `-`, `*`, `/` and `%` (for modulus). A sample expression would be:

```
$a + 5 * 10
```

If `$a` is 3, this expression evaluates to 80, not 53. That's because the `+` is done before the `*` since it's further to the left. The `-` operator can also be used as a negation operator, but only as the first operator in an expression. So `$a = -$b` is fine, `$a = $b * -$c` doesn't work.

With parentheses, then the expression:

```
$a + ( 5 * 10 )
```

will then return the value 53. Parentheses sometimes require a spaces before and after them so it's a good idea to always insert them.

For strings, the `^` operator is used for concatenation. So:

```
“hello” ^ “there”
```

evaluates to “hellothere”.

Type changing (casting) is done automatically based on the type of the first variable in the expression. So if `$b = 1` and `$c = “5”`, then `$b + $c` will be evaluated as 6 while `$c + b` will result in an error (because `+` can't operate on strings). But `$c ^ $b` will result in “51”. You can force the type of a variable to a number like so: set `$a = 0 + $a` or to a string: set `$a = “” ^ $a`. Note that the literal has to be the left most term.

Booleans values are also supported. When converting values, 0, “false” and “” will convert to false, everything else to true.

Boolean expressions are also supported. The 6 standard comparison operators are available: <, <=, ==, !=, > and >=. In addition & and | are the boolean AND and OR operators. You can now combine the boolean expressions with the use of parentheses. This is wrong:

```
$a > 0 & $b < 0
```

as it will attempt to do the AND operation between  $a > 0$  and  $b$ . The correct way to do it :

```
($a > 0) & ($b < 0)
```

The only operators that work on objects are the == and != ones which are used to determine if two objects are the same. However, objects in a boolean context evaluate to true if they are defined, to false if they are not (set to null in the C# code). So if you have a character P, doing if(P) { } ensures that P is defined.

The output of most expressions will be stored in another variable. You do this using the set command. This looks like:

```
set $varname = <expression> ;
```

The **set** keyword is now optional!

Expressions can also contain the results from function calls:

```
$a = $b + &call();
```

Individual container entries can also be included in expressions:

```
$a = $b * $c[$d];
```

and can be similarly be set:

```
set $c[$d] = $c["x"] + $c["y"]
```

The container index must be a literal or a variable. It can not be an expression or a function call. Note that if using a variable as an index, if the variable is a number, it will be converted to an integer (rounded) and then to a string.

## Callables

Callables can be used in expressions if they return a value. Callables can have any number of arguments, as in:

```
set $a = &call_name(P, "hello", 3, $x);
```

```
&do_something("my_thing");
```

Note that while arguments can be expressions, they can't include other calls. So

```
&foo($a + $b);
```

is allowed,

```
&bar(&foo($a))
```

is not.

To call a callable in another module, prefix it with the module name:

```
&mod_name::func_name();
```

There are a number of built-in function calls that can be used just like this. Check the cheat sheet file for a complete list of these functions. Built-in functions never have a module prefix. You shouldn't use these names as names for your own functions. It may be allowed and may work in some cases, but it's not guaranteed to.

This makes it impossible to override built-in functions on a global level. Whether this is good or bad is not clear at the is time.

Functions are defined with the function keyword in a pretty standard format:

```
function &<func_name>(<arg_list>) {  
<block>  
}
```

The variables defined in the argument list (formal parameters) are declared implicitly and do not require a separate declaration. Objects are passed by reference, all others are passed by value. Containers can not be passed at all. (They can be passed to built in functions, though). Function signatures are not validated at parse time. However at run-time, if the number of actual parameters doesn't match the number of formal parameters, an error will occur. Similarly if one of the actual parameters is of a type incompatible with the formal parameter, an error will also occur.

In addition to functions there are 3 other types of callables: actions, interactions and events. These will be discussed further in a later section.

## **Object Properties**

The object variables also support a number of properties, accessed via a syntax like:

```
$age = P.age
```

This sets the variable \$age to the age of the character stored in P. Properties can be used in the same places where indexed containers can be. Essentially anywhere except as container indexes. Most properties are read-only, but there are a few that can be read/write. The cheat sheet contains a the reference info for all available properties for each of the five object types.

Properties are a more efficient and faster way to access data from objects without having to call built-in functions. However, they only work on one object at a time.

## **Control Flow**

The TFL language does not have a lot of control flow statements. It basically has a loop and an if-then-else statement (with 2 special variants). There's a special case **foreach** loop as well.

The if statement is pretty standard. It looks like:

```
if (<expression>) {  
<block>  
}  
elseif(<expression>) {  
<block>  
}
```

```
else {  
<block>  
}
```

Like in Perl, the braces are required even if you have a single statement in the block. The **elsif** and **else** sections are optional, and any number of **elsif**'s are allowed. When evaluating the expressions, any value is converted to boolean, so 0 values, empty strings and null objects are evaluated as false.

The **ifall** and **ifany** statements have been deprecated now that compound boolean expressions are supported with the use of parentheses.

One thing to keep in mind with boolean expressions is that the entirety of the expression is evaluated every time. For example:

```
if($a & &big_func_x()) {  
}
```

the **&big\_func\_x()** function will be called and executed every time, even when **\$a** is false! So for performance reasons, it may make sense to write this instead as:

```
if($a) {  
    if(&big_func_x()) {  
    }  
}
```

The **while** loop is also straight-forward:

```
while(<expression>) {  
<block>  
}
```

The block statement is evaluated while the expression is true.

Finally the **foreach** statement:

```
foreach $var in $cont {  
<block>  
}
```

This one is a bit more complicated. This will run the block statements for each entry in the **\$cont** container. If there are no entries in it, it won't run any code. Note that the **\$var** variable is implicitly declared in the **foreach** statement and does not need a separate declaration. The value of **\$var** is every value in the container, not the keys. So this is best used on containers that represent lists. (A container that represents lists tends to have the ordinal numbers 1, 2, 3, etc. as keys).

One limitation of **foreach** loops is that they can't be nested inside other loops. If you need to do this, you will need to put the inside loop in a separate function. This is a run-time error so you will not get a notification about this until the code is executed (during the second iteration of the outside loop).

Also related to control flow are the **return**, **break** and **finish** statements. The **break** statement is used inside a **while** or **foreach** block to exit the block immediately. The **return** statement is used to

terminate the execution of a callable. It accepts an expression which is evaluated and the value returned to the caller. There's not checking to make sure that callables used as functions actually return a value. If no value is specified, a value of null or 0 or false is returned. Similarly a callable used as a "procedure" (i.e not in an expression) which returns a value, will have said value ignored.

Finally, the **finish** statement terminates the entire stack execution. It returns control back to the platform code. You can use this to abort an action deep inside a function. This should be rarely used.

## **AIE Callables**

Earlier we mentioned callables and mostly showed how one type of callable, functions were used and defined. The 3 additional callables are similar and are referred to as AIE callables. AIE stands for Actions, Interactions and Events, which are three types of scenes that can be run. They all have a very similar way to declare them, so before we get to that we will look at the differences.

Actions are scenes that are driven by a single character. This character can be the MC or it can be an NPC. Examples of actions are: resting, going to another place, buying more inventory, etc. The unique aspect of actions is that they take a single character as an argument. The action itself can involve more characters, but the action code is responsible for finding those characters and bringing them to the scene. Actions are triggered by the one character argument.

Interactions are scenes that involve two characters. This could be an MC-NPC scene, and NPC-MC scene or a NPC-NPC scene. Examples involve two character having a chat, a character fighting another one, or a character purchasing something from the other. Interactions are unique in that they take two characters as arguments. The first character is the driver or protagonist of the scene normally called the "actor", is the one who triggered it. The second character is the "target" of the interaction.

Finally there are events. Events are things that happen but are driven by other factors than characters. Events can bring chracters into it (and usually must to do anything worthwhile), but they are triggered via conditions that happen outside the characters. For example, an event could be a disaster, or something we want to happen without a character being responsible for. Events take no arguments.

Although rare, it is possible to call AIE callables from other callables, including functions.

AIE callables are different from functions in that they have a preamble in their declarations. AIE callables are triggered by the AI and this preamble is used to determine when to do that, and to do so as efficiently as possible. The biggest impact to performance in the system evaluating all the possible AIE callables and figuring out which ones to run for each character.

The preamble consists of 4 sections, 3 of which are single lines so maybe calling them sections is too grandiose. These sections are:

- The details section
- The init section
- The allow section
- The aiprob section

A sample AIE callable definition looks like:

```
action @action_name(X)
    details "Action Name >", 15, 5, 30, "owner_only"
    init {
```



```

        <block>

    }
    allow <expr>;
    aiprob <expr>;
{
    <block>
}

```

The four sections of the preamble must always be present and must be present in that order. They also must start on their own line. Note that the last two require a semi-colon at the end.

The **details** section defines 6 parameters and it's the first thing that's used to determine whether a given AIE callable is valid at a given time. The six parameters are: AIE callable name, AI chance, timeout, sort value, execution flags and taboos. These values are used to quickly eliminate as many callables as possible from consideration. The name is used on Actions and Interactions to show it to the player in the menus. If it's blank the callable is not available to the player. In general, if the callable name ends with a ">" it means that executing that Action or Interaction will move the time to the next period. It's not always perfect as choices made inside the callable may sometimes determine if time passes. The last two don't need to be specified. (But if you want to specify a taboo without the execution flags, you need to set the flags to a empty string).

The AI Chance tells the system the probability that this callable should be considered at all. Normal values are 10-20. A value of 15 means that it will only be considered 15% of the time. With 6 ticks per day that AIE would still be considered almost once every day, for every character. Use this first to adjust your actions or interactions so that they happen more or less frequently.

The time out tells the system that once it runs a particular callable for a character not to run it again for this many days for that same character (or globally in the case of events). Set it to 0 to have no timeout. The timeout does not affect the MC, only NPCs. The timeout has an effect if the callable is triggered, not if it ran successfully or to a particular point.

The sort value is used to sort the items in either the Actions or Interactions menu. The higher the value the higher it will appear in the list.

The execution flags is a string literal which has a space-delimited list of conditions that all need to be true for the callable to be triggered. These flags are things like "owner\_only", "at\_park", "during\_workday" which make the callable available if the caller is a club owner, is at the park and the time is a work day. These are to be crude ways to weed out some callables as quickly as possible. A complete list of these flags appears in the cheat sheet. Unlike the ai chance and the time outs, these flags do affect the MC. If the MC does not meet the conditions required by the flags, the action/interaction is not shown.

Finally, the taboos is a string literal which has a space-delimited list of taboos. A taboo is a type of content that some users may not want to see. In the config.dat file, one can specify a number of these taboos. Then the AIE developer can qualify it by specifying which of the supported taboos appear in the callable, if any. Then the system will make that callable not be accessible for that game. Currently there are four supported taboos: incest, non-consensual sex (noncon), pregnancy and racism. For example, if you set the "taboos" entry in the config.dat file to "incest", and then create a callable and put "incest" in its taboo list, then that callable will not fire during that game as part of the AI selection

process, nor will it be accessible to the MC. It can however fire if called from another callable not designated as such. For that reason, this is meant as a setting for top level, initial callables.

After the details, the second section is the **init** section. This has the format of a simple block. The purpose of this is to have the code necessary to calculate the values for the allow and aiprob fields. Any variables needed here need to be declared as well. This code should be as efficient as possible as it will be executed a lot. Put the faster tests first, as well as the ones more likely to fail. Variables declared in this section do not need to be declared again in the main callable block. But the values will not transfer.

For interactions, the init section needs to support two situations. There's the standard case where the init() function returns true or false depending on whether the interaction is valid between the actor and the target. But it also needs to support a more generic case, where the target is null. In this case the init needs to return true or false to determine if the interaction is valid for this actor at all, regardless of the target. This improves performance as it can quickly rule out an interaction for a given character no matter who the target is. Failure to handle the situation where the target is null will result in the interaction never running.

**IMPORTANT:** One should never, ever, put commands that change global values (either flags or object properties or call built-in functions that change data) in the **init** section. The init section will be executed a lot of times to determine if the interaction will be run.

The **allow** section takes an expression that's evaluated. If it evaluates to false, the callable is not triggered. In general, the init section will have code that sets variables which then can be tested in the allow section. The allow section applies to the MC as well. If it evaluates to false, it will not be listed as an option for the player.

The **aiprob** is used to further decrease how often a callable runs. If allow indicates that its ok to run, the aiprob then figures out whether to run it that time or not. The difference between this aiprob and the AI Chance in the details section is that this aiprob expression can use data specific to the character(s) involved in order to make that decision. Use this to incorporate the NPC's personality in determining how likely someone is to do an action or interaction. The aiprob never affects the MC. Note that you need semi-colons after the allow and aiprob expressions, but not after the details or init one.

Note that in the rare event that an AIE callable is called directly from another TFL code callable, the AIE callable will always execute and the preamble conditions (either on the details line or the allow\_it/ai\_prob checks) will not be tested.

Once the callable has determined that it can be triggered it's then put on a list and one callable is selected from that list to be executed. At that point the callable is considered as having run for the timeout to be in effect the next loop. If the callable did not use all of the character's moves, the character is given another random callable from the list to execute.

## **Decisions**

The **decision** statement is a very powerful construct in the language and it's the the most unique aspect of it. It is used to get a character to make a decision. If the character is the MC, it shows a dialog to the player with some choices and awaits a decision. If the character is an NPC, the system AI, using the parameters provided in the **decision** statement and taking into account the character's personality, makes the decision.

The **decision** statement thus contains 3 types of parameters:

- Information for the MC about what's going on

- A list of choices that can be made
- Information for the platform AI to make an NPC decision
- Additional info for the MC to explain the consequences of the choice or other matters related to the decision.

The middle one is always required. The first is not required if the decision will never be shown to the MC and the third is not required if the decision is only shown to the MC. In general however one should write decisions so that they apply to both the MC and NPCs.

The format of the **decision** statement looks like:

```
set <var> = decision <char> [,<char list>] {
    [title <loc_spec>]
    [<body_spec>]
    <choices_spec>
    [hint <hint_id>]
}
```

At least one character is required in the decision statement. That will be the character making the decision (called the protagonist). Up to 5 additional characters may be listed, separated by commas. Those 6 characters will be the ones involved in the decision. If the protagonist is the MC, then the additional characters will be shown in the display. The protagonist will be character 0, the next one will be character 1, etc. Character 1 will be shown as the main image on the right. Character 2 will be shown as the secondary image, which will be slightly smaller and offset to the left. Characters 3-5 will then be shown as small thumbnail headshots on the bottom, left to right. In order to force a character to a thumbnail spot without using one of the two larger images, specify “-” for the larger images instead of a character variable

The title takes one argument which must be a localization string literal without a replacement value. No variables or expressions are allowed here. Localization strings will be discussed further on. The title line is optional, but only one is allowed in a decision. If present, the title is displayed at the top of the decision dialog.

And the <body\_spec> consists of 0 or more body lines. The body lines can be of one of two forms:

**body** <loc\_spec>

**bodyif** <bool\_expr>, <loc\_spec>

The standard **body** line takes one argument which is a localization string with or without a replacement value. The replacement value is specified after the key separated by a tilde. See the Localization section for more info. The **bodyif** line takes two arguments, the first of which is a boolean expression. All the **body** lines, and all the **bodyif** lines that evaluate to true are concatenated and displayed as the body of the dialog in the order that they appear in the statement block. When displaying them they are separated by spaces, not newlines.

The <choice\_spec> consists of a series of choices for the decision. The protagonist will choose one of these, so at least one valid choice must be given. The choice can be specified via either the choice or choiceif line. At least one choice line is required. Then each choice can have any number of impact modifiers:

**choice** <term>, <expr>, <label\_spec>

**choiceif** <term>, <bool\_expr>, <expr>, <label\_spec>

**impact** <simple\_term>, <attr\_key>, <multiplier>

**choose** <choose\_type>

Here's where things get a bit complex. The <term> must be either a string literal or a single variable. Expressions are not allowed. However a function call or variable property or hash element are allowed. This is what identifies the choice and what will be returned by the **decision** statement if that choice is selected. When **choiceif** is used, the choice will be made available if the <bool\_expr> evaluates to true. When using **choice**, the choice will always be available. The <label\_spec> must be a string literal representing a localization string or a variable containing the actual string to be shown. The variable must be a scalar, an indexed container is not allowed. Note that the replacement value logic from the body lines is not allowed. Use string literals whenever possible, as this allows additional functionality as will be explained below.

The expression <expr> (the 2<sup>nd</sup> argument of **choice** or the 3<sup>rd</sup> of **choiceif**) is evaluated as a float and it refers to the weight the AI will give to this choice. For MC only options this value can be anything but should always be higher than 0. The impact modifiers will be used to automatically modify the weights based on the protagonist's attributes. The protagonist's value for the given attribute will then be multiplied by the multiplier and this product added to the weight. Again, no impacts are needed for MC-only decisions. For the AI if the total value for a choice evaluates to 0 or negative, it will not be considered.

The first argument of the **impact** line must match a previous **choice** or **choiceif** line. It doesn't have to be the immediately preceding one but it needs to be after it. There may be multiple **impact** lines for the same **choice** or **choiceif**. However, **impact** lines are only allowed when the **choice** or **choiceif** it corresponds to was identified with either a literal or a scalar variable. If it was identified via a function call or hash element or variable property it will give a compile error. That's because the matching it's done at compile-time. On the impact line, the attribute key and multiplier must both be literals. No variables or expressions are allowed.

The choose line is rare, and it's used to define how the choice should be made. The valid values are **best**, **weighted**, **strong**, **random** or **same**. The default is **weighted**, which means that the character's personality will be used to score each choice and then pick one at random using the scores as weights. So if there are two choices and one scores 75 and the other scores 25, it will pick the first one 75% of the time. The **strong** option is just like **weighted**, except the weights are squared, so the higher weight options are much more likely to be picked (i.e. it decreases the effects of the RNG). Note that **strong** works best for 2 options that add up to 100. When **best** is specified, it will always pick the option with the best (highest) score. If **random** is selected it will pick one option at random, and the scores will be ignored. The last option, **same** is the same as weighted but it will ensure that the same choice is selected each time on the same day (i.e. the character will always make the same choice when presented with the same decision multiple times in the same day). Note that same is not fully guaranteed to work. In particular, some subtle differences between the way the decision is called may cause it to be re-evaluated. Also, this info is not saved, so saving a game, and then loading it resets all such decisions.

In the event that there are no valid choices, the decision will return null (or ""). One should avoid this. On the other hand, the parser does not enforce a maximum number of choice statements. At run time only the first 10 choices will be displayed to the MC. The AI will have the full set to choose from.

Finally, the optional **hint** line provides an additional id that identifies the hint for this to be displayed if the game is showing hints. This must be a string literal that will be converted to a localization string

using the format "hint.<hint\_id>". It can't be a variable and no replacement strings are allowed.

Here's a sample decision:

```
set $ans = decision A, B, C {  
    title "my.title"  
    body "body.line.1"  
    body "body.line.2"~$value  
    bodyif $x > 0, "body.line.3"  
    choice "yes", 100, "yes.opt"  
    impact "yes", "will", 0.5  
    impact "yes", "soc", 0.25  
    choiceif "maybe", $decisive < 20, 50, "maybe.opt"  
    impact "maybe", "will", -0.5  
    choice "no", 100, "no.opt"  
}
```

For 2.0 there's one additional decision element, and that's the **prompt** element. The prompt is used to ask the user to enter a text string, which is then loaded into the variable. The format is:

**prompt** <expr>, <label\_spec>, <def\_val\_spec>

The prompt field takes three arguments: the max size of the text to enter, a localization string specifying the label to show to the left of the input field and the default value to show. The max size should evaluate to an integer. Note that this only prevents the user from entering strings longer than that value, it does not change the size of the input box. The default value can be a localization key, a variable or a literal. To specify that the default should be blank, use the localization key "null" (in quotes). Specifying an empty string will show a pair of quotes as the default. Note that specifying the default via a literal will result in a warning. It may also collide with an localization key. So it's always recommended to either use a localization string or a variable to specify this value.

Note that the **prompt** field only works for the MC. If the AI is given as the protagonist of the decision, it will throw an error and the function will be aborted. Also, only one **prompt** field is allowed and it can not be included in decisions that have the standard **choice** options. This option is for the very specific purpose of having the user enter a single string value. The value may be converted to a number explicitly via the &int function or implicitly in an expression.

The elements of a decision must be specified on separate lines. But they can be specified in just about any order. Normally one would do title, body, choice, impact but the order doesn't have to be that way, other than the impact lines need to come after their respective choice specification. Finally it's worth noting that there should be no semi-colon after the closing brace of the decision. As a general rule, don't put semi-colons after a }. This also comes into play in the init section of an AIE callable preamble.

Since the decision statement displays a dialog to the MC, it is also used to display information to the player, even though no decision is required.

## Object Selection Interface

In addition to the decision command, TFL offers a second way to get user input via an interface while also allowing the AI to make the selection for NPCs. This is the **select** command which is used to allow a character to choose among a number of things via some criteria.

The format of the command is:

**set** <var> = **select employees** | **acq** <char>, <crit\_expr>;

At this time the only options is to select employees or acquaintances, but in the future there may be more options. (The future is here now, additional options are **workers** and **performers**.) This is specified as a keyword following select. Note that this is not a string literal (no quotes) and it can not be the result of an expression or a variable. It takes two arguments, one is a character variable which indicates the character that's making the decision and the criteria used to make the decision. If the character is the MC, the criteria is ignored. The criteria is a string literal or expression that evaluates to a list of things to consider. This list is still under development but here's the current list: attraction, opinion, rev\_attraction, rev\_opinion, job\_skill, salary (in 000's, low values better) or an attribute key. A '-' in front of the attribute multiplies it by -1. All these elements are weighted the same. To make one count double, you can list it twice. This can be a bit performance-intensive, so try to limit these to a very short number.

In addition to the list of criteria, there are three additional “commands”, one of which can be included to tell the system how to make the decision. These are “random”, “weighted” and “best”. If “weighted” is chosen, it will pick one option at random using the evaluation of the remaining criteria as the weight. I.e if there are two potential characters and one's criteria evaluates to 20 and the other to 30, then there will be a 40% chance that the first one will be chosen and a 60% chance that the second will. “best” works by always picking the character with the highest value. Finally “random” means just picking one character at random without taking into account the rest of the criteria.

With the select interface, it is possible that no selection is made. This can happen if there are no valid choices (i.e. the owner has no employees or is not even an owner) or if there are choices and the MC exits the list dialog without making a selection. The code that follows it needs to handle this situation.

In addition to **employees** and **acq**, three other options are available: **workers**, **performers** and **clients**. Performers will consider employees that work as performers only while workers will consider employees that are not performers. Clients considers characters that can be club clients, so non-club employees or owners.

## Scheduler

The scheduler is a feature of the platform that allows to trigger callables at a time in the future. It's how complex scenes and story lines can be implemented. For example, if two characters arrange to have a date the next few days, the scheduler is used to trigger the date scene at the appropriate time. When specifying a function to be scheduled, the date when it runs as well as the valid time periods when it can run are specified. The function will then be run at one of the valid time periods on the given day. Internally I use the term “tick” to refer to a time period, so the two terms are interchangeable.

The syntax for this command is:

**sched** <expression> **days** | **months** | **years**, <tick\_spec> , <callable\_id>( <arg\_list> )

The expression is evaluated as an integer. Normally one would specify it in number of days, but specifying it as number of months and years is also supported. The tick specification is a literal, either a string of a tick type or a number. String values are things like “anytime”, “daytime”, “nighttime”, etc. A

full list appears on the cheat sheet. Or it may be specified as a bitmap integer, where 0x01 stands for Tick 0, 0x02 stands for Tick 1, 0x03 stands for Tick 0 and Tick 1, etc. The literal needs to be specified in base 10 unfortunately. Note also that only the time period in the string tick specification matters. So if using something like “weekend”, it's essentially the same as “anytime”. If you want to run it on a weekend, you need to set the number of days correctly. You can set the number of days to 0, but in that case the tick specification must be for a specific tick, after the current one.

Finally the callable specification is a standard, fully-qualified callable reference, including parameters. The parameters are evaluated when the function is scheduled. An example:

**sched** 14 days, “nighttime”, &club::visit\_club(club:C, char:P);

This will run the function &visit\_club (defined in module club) in 14 days during one of the night time periods, passing the club pointed to by C and the character pointed to by P.

Why is the scheduler a TFL syntax element and not just a built-in function? The reason is that passing the function to be scheduled as well as all the arguments would have required having to evaluate a string and interpolate variables in it at run time. This would have not only been tricky code to write but also inefficient. I rather have the parser handle it at compile time. The parser since then has become more sophisticated that it could be done as a function call but the keyword approach was already in place.

Sometimes when a callable is scheduled, an entry is added to the MC's calendar. This is controlled via the data/calendar.txt file and will be discussed later on. Calendar entries can also be added via a built-in function when the automatic method is not sufficient.

## **Pragmas**

Pragmas are essentially compiler directives that appear in the TFL code. They live sort of outside the regular program flow. Pragmas take the form:

##! <pragma> <args\_list>

At this time we have 4 pragmas:

- **DEBUG** pragma. It accepts an argument of 0 or 1. If the argument is 1 and you are running in dev mode, it writes a lot of debugging info to the console and the game log file. This includes the values of all callable arguments and their return values. The pragma is context sensitive and it takes effect exactly where it appears. It will only take effect if the statement immediately preceding it executes. So if you put it inside an if block, and the if expression is false, then the **DEBUG** will not be activated. The debugging is turned off automatically at the end of the callable (not the block!), so no need to having to manually do that.
- **TEST** pragma. This is used for scripts used for testing the parser and compiler. It can be ignored for the purposes of creating mods and the published version of the game has this functionality turned off.
- **MAX\_STMT** takes one argument and its used to change the number of TFL object code statements that the platform will allow the callable to run. The default is 10,000 statements which should be enough for most operations. But if one needs more, the **MAX\_STMT** pragma is there to change the limit. It's worth noting that the limit is per callable. So a callable may run for 5000 steps, then call another function that runs for 4000 steps and then come back and run for another 3000 and it will be fine. The purpose of this is to avoid infinite loops locking up the game or badly written code slowing things too much. The counts is of object code statements.

On average, one line of TFL code turns into 5 object code statements. Mostly this is only needed on callables that iterate thru the entire list of characters in the game or all the relations a character has.

- **Multi-line comments.** This one works a bit differently, but you can comment out a block of code using the { and } pragmas. This is only to make code not be compiled into the game. It is not meant to be used for comments. For example:

```
##! {  
function &abc($b) {  
    var $a;  
    $a = 100 + $b;  
    return $a  
}  
##! }
```

In general is best to put pragmas at the top block of a function/callable, after the variable declarations but before any executable statements. In particular, putting them after a conditional statement or a loop will likely not work. The multi-line comment can be placed outside a callable code.

### **Misc Stuff**

There are a few remaining keywords in the TFL language that haven't been addressed. One is the **debug** statement. It takes the form:

```
debug <expression>;
```

The expression is evaluated and the result is written to the debug log (and console) if the program is running in debug mode and debugging is turned on via the debug pragma. The debug statement takes a string expression as an argument. This can be used to print variables and object properties as necessary, like:

```
debug "Got here: " ^ @A.name ^ " with value x=" ^ $x;
```

## **2. Character Image System**

This section is essentially a copy of the *SCW Character Image System.pdf* that can be found in the game's home directory. The information here may not be as up to date as on that file so it's recommended that you read the stand-alone file instead.

Character images are AI-generated images used to represent the characters in the game. This system is built to allow for ease of adding images after the game's release by both the developer as well as community modders.

Images are stored in the *characters* subdirectory of the SCW data directory. For convenience, the official images are grouped in directories with 50 characters per directory. So for men, they are in the directories *scw\_males1*, *scw\_males2*, etc. And for women its *scw1*, *scw2*, etc. There's also a *Defaults* directory which contains some special character images. Community-created images should be stored in directories with names that don't start with *scw*. However, there's no minimum or maximum number of images that may be stored in a particular directory.



Each standard character has 7 or 13 images associated with them, depending on their gender. Male characters have 7 and female characters have 13. Each image is called a “pose” although they are all usually the same or similar pose. What's different is what they are wearing. (Initially it was supposed to be an image for different poses and then clothing was going to be a different layer but that proved impractical for now. But the term “pose” was too ingrained and stuck.)

The 7 poses for men also apply for women. Women have 6 additional poses. The 12 poses are:

1. Nude [nude]
2. Topless [tl] - (women only) wearing panties or a bikini bottom usually)
3. Underwear [uw] - boxers/briefs for men, panties and bra for women usually)
4. Swimsuit [ss] - (women only)
5. Fun [fun] – workout or lounging around clothes, like shorts and a t-shirt
6. Casual [cas] – casual clothes, jeans and polo shirts/blouses
7. Business casual [bc] – nicer than casual, slacks and long sleeve shirt for men, skirt and blouse for women
8. Business [biz] – business suits
9. Strip 1 [s1] – (women only) entertainer outfit 1
10. Strip 2 [s2] – (women only) entertainer outfit 2, more revealing than strip 1
11. Strip 3 [s3] – (women only) entertainer outfit 3, more revealing than strip 2
12. Pregnant [preg] – (women only) with a swollen pregnant belly.
13. Headshot [head] – a thumbnail headshot

The labels inside [brackets] are the codes that identify each pose in the filename. All the meta data about an image is stored in the filename itself. There's no master data file that provides this info. This is to make the system as distributed as possible. But it means that the naming scheme is very strict.

Note that while all the characters in the official SCW image packs contain their full allotment of poses, it is not necessary that all poses be defined. However, a bare minimum of the headshot, casual, underwear and nude is required for proper operation. If those poses are missing the game will continue to run but the text and the images may not be in sync and default images may be plugged in. If the headshot image is missing, the corresponding poses are ignored and not loaded. Missing images and other issues reading these files will be logged into the output log file.

There are some special characters which are used for some particular storylines that do not have all their poses because they can only be interacted in specific ways. These character images are stored in the *Defaults* subdirectory. For example, the policeman that shows up every once in a while is one such special character. He won't appear or interact other than on scenes that specifically call for him in his role as a cop. As such we only need the one image of him in his policeman uniform.

## Character Image Details

Other than the headshots/thumbnails, the images have a size of 512x800 pixels. Different sizes are allowed, but the system will fit them in spaces of that size at most. So any larger than that and it's a waste, any smaller and it would be lower quality. Sometimes the images may be shown at a slightly reduced size, but using the same aspect ratio. If using smaller images than the default size, try to get the

aspect ratio as close as possible to avoid distortion.

Headshots are 120x160. In general they are the same scale as the full images. The size itself doesn't matter much as long as it follows a 0.75:1 aspect ratio. Although again, making them larger is just wasteful.

Non-headshot images should be full body or close to it. And the vast majority of the images are people standing facing the camera. A few are from the side and tiny number are from behind with the person looking back at the camera. There are also a small number where the character is sitting, kneeling or squatting. The AI image generation is much better at front-facing images and making those others require much more trial and error.

Images must be in PNG format with their background being transparent. This unfortunately makes them much larger size-wise than a corresponding JPG but transparent backgrounds is a necessity. Only files with a .png extension will be considered. Any other files in those directories will be ignored.

## Character Image Filenames

The headshot is the main image of each character. This provides the most info about the image and the characters to which it can be attached to. The format of the headshot image files is *modkey-id-reqphys-optphys-imgphys-x-head.png*. The filename has 7 components, separated by dashes. Each of these components is explained here:

- *modkey*: This is short code identifying what package the image belongs to. All the basic (vanilla) game images will have the modkey "scw". Anyone that creates images may do so using a hopefully unique modkey (and different from 'scw'). To make things easier to manage, the modkey should also be used as the start of the directory name where the images are. This key must be added to the image\_module config value entry for the game to recognize it. See the section below for more details.
- *id*: This is a 5-digit, zero-padded number. Each character image set should get it's own unique number. Internally images are identified by the concatenation of the modkey and the id. If the system finds two or more files with the same modkey + id, it will only save the last one it loads. Each mod can have whatever numbering scheme they desire since the numbers can be shared between mods as long as they have a different modkey. For the base game images, male images begin with a 1 (10000 and above) while female ones start at 0. Special characters begin with a 9.
- *reqphys*: This is where things get a bit tricky. This is the first of the 3 components that match the image to the character physical attributes. This first set is the required attributes. It's a 3-letter code that indicates the gender, ethnicity and age group of the character. Only characters that match those attributes can use this image. This way a black 40 year old man will never get the image of a asian 20-year old woman. Gender is specified with *m* or *f* (obvious), age is a number 1-5 (age groups 18-24, 22-31, 28-42, 38-51, 48+) and ethnicity is one of *w*, *b*, *h*, *a* or *r* (white/caucasian, black, hispanic/latin, asian or middle-eastern). Note that the age groups do overlap. For example 'f3h' refers to a 30-something hispanic female.
- *optphys*: This the second of the physical attributes components. These are optional or suggested ones. When looking for an image for a person, the required parameters (gender/age/ethnicity) must be met. These one don't have to, but the system will pick the one that comes closest to the person's attributes. There are 5 attributes here: height, body shape, hips/butt size, breast/penis size and skin tone. For height the values are *t*, *m*, *s* (tall, medium and short). Note that the

images don't do a great job of conveying height due to the lack of context in the form of other objects so this is usually the least important. Body shape is one of *s*, *n*, *c* or *f* (slim, normal, curvy and fit). Note that most normal images are fit by any common definition of fit, but fit here is usually very muscular characters, not just in good shape. Hips/butt size is one of *s*, *m*, *l* (small, medium and large). And breast/penis size is one of *s*, *m*, *l* and *h* (small, medium, large and huge/unrealistically large). Obviously the breast applies to women and the penis applies to men only. There's not support for women with dicks in this game at this time, if and when we add that we will need to change this somehow. Finally skin color is pretty generic, with values being *l*, *m* and *d* (light, medium and dark). Almost all caucasian and asian characters have light skin, black ones have dark skin and latin and middle eastern have medium skin.

- *imgphys*: These are the physical components that don't play a role in game play other than to add flavor. So a character will inherit these values from the image when they are assigned an image. The 3 attributes in this group are: hair color, hair length and eye color. Hair and eye color values are *l*, *m* and *d* (light, medium and dark) while hair length is *b*, *s*, *m* and *l* (bald, short, medium and long).
- *special*: The 6<sup>th</sup> field is 'x' for all standard characters. For special characters we use this field to define their specialness. For example, the cop will have 'police' as the value in this field.
- *pose*: The 7<sup>th</sup> field is always 'head' and specifies that this is the headshot.

Non-headshot images have shorter filenames, to avoid duplication from the headshot. The file name format for these is *modkey-id-rev-pose.png*. The first two fields *modkey* and *id*, must match that of a headshot image and it's used to link all the pose images for a given character.

The *rev* field indicates how revealing is what the character is wearing. This is a somewhat subjective value that goes from 0 (totally modest) to 12 (very immodest). The general rules about this field are: most underwear poses are 3, although sheer ones may be higher. Standard nude poses are given a value of 9, although spread open vaginas and erect penises would be higher. Topless poses (for women) are a 6. Most swimsuits are 2, while sexy clothing but not revealing much are 1.

Some of the older images do not have the value in the *rev* field. In this case, the default revealing value for that pose is used. For example, business, business casual, casual, fun have a default revealing value of 0. Swimsuit is 2, underwear is 3, topless is 6, nude is 9. This practice has been discontinued and new images should explicitly set their reveal value.

The average revealing factor of the main poses is combined to create an overall lewdness of the character image set (main poses are fun, casual, business casual and business). The system will try to assign images sets to characters with a lewdness attribute that fits the image set. This has more weight than any of the optional physical attributes (height, body shape, etc.). Sometimes the character physical attributes will be modified to better fit the image, but the lewdness will not. If you make images that are too lewd, they may never get used.

Finally the pose field indicates the pose itself using the codes listed above, like 'biz' for business and 'tl' for topless.

## **Recognizing New Images**

After creating the new images, you need to tell the game about them. To do this you need to add the modkey you used for your images to the *user\_config.dat* file like this:

```
image_modules string scw,modkey
```

Where *modkey* is the prefix you used for your images. The order doesn't matter. Make sure to include "scw" otherwise the standard game images won't show up and the game likely won't work.

### ***Additional Images***

Starting with version 1.01, the system will allow multiple images for each pose, as long as they have a different revealing value. For example, you can have a character have two casual images, one with a z-value of 0 and another with a z-value of 2. The game will then pick which image to use based on context and the character's personality.

### ***Pregnant Images***

Starting with version 1.07 the system supports images for pregnant characters. These will have a pose code of "preg". These will behave differently in that the clothing should be somewhat generic. Different types of clothing can be used by using different revealing values.

Now creating a pregnant version of every image is completely impractical, so the rules to when to use the pregnant images will change. If the character is deemed to be pregnant enough to be showing, and there's at least one pregnant image available it will use the pregnant image available closest in revealing value to the original non-pregnant image it would have shown. Otherwise it will use the non-pregnant one. The difference in reveal value must be 4 or less. For example, if there are 2 pregnant images, with reveal values of 1 and 9, and the system wants a topless image (normally reveal=6), then it will use the nude pregnant image. If it wants a standard casual image (reveal=0), it will use the one with reveal value of 1. If it wants one with reveal value of 5, it will use whichever it found first, since they are at both a 4 point difference.

### ***A Few Additional Tips***

Some additional tips for better results:

1. Nude images should have a revealing value of at least 9. Setting it to less than that will have the game not-recognized them as being fully nude in some scenes (like the stripper not undressing scene). All non-nude images should have a z-value less than the nude image. Otherwise you may get a situation where a nude character undresses to a pose wearing some clothes.
2. Topless images should be at least 6. For women, a z-value of 6 would indicate that they are topless. The z-value is what's used to determine when someone is dressed legally, so again, it's important that the images match the z-values in this respect.
3. Images in soft light (i.e. without areas in direct sunlight/shadow) work best.
4. Removing the background of an image is tricky and often leaves a halo around an image. The halo annoys me, specially around the hair, but attempts at removing it tend to make things worse.
5. Images should be facing the camera, it's ok if they are from the side or behind, but it works better if they have their heads turned towards the camera.
6. When creating images sets, resist the temptation to make all images very revealing. At least one of the 4 daily wear images (fun, casual, bc, biz) should have a revealing value of 0 or 1. Otherwise some characters may not find anything to wear and the results could be weird. Like wearing a swimsuit to work.

7. When running out of images and the system can't find an appropriate image for a character, it may reset some of the character's physical attributes (height, body shape, hip size, breast/penis size only) in order to better match an image set. Gender, age, ethnicity, skin color and lewdness will not be changed no matter what.
8. There's no good way to override an existing image. You would need to actually either delete all the images associated with it or replace them. Creating an image set with the same name in another directory means that the last one read in will be the one taking precedence. At this time there's no way to enforce any precedence, although that will change in the future.
9. The images in the *Defaults* directory can be overridden, but they should not be deleted. Deleting them will result in no image being displayed. This directory contains two types of images: the masked images and images for special-purpose characters that can only be interacted in specific scenes. The masked images are used when there's no available image to show for a character (i.e. we ran out of images for that particular gender and ethnicity) or when you have a contact that you haven't yet met face to face (for example, your sibling recommended someone for an interview). Until you met them face to face you won't know what they look like and the masked image will be used.
10. If you do add your own images, when you restart the game, check the log file (typically under (C:\users\<YOURNAME>\AppData\LocalLow\Total Fluke Studios\StripClubWars) for any errors. If it was unable to load any of your images, there should be an error towards the top of the log file. Images are only read in at game start, so you must restart the game to see any changes.
11. When creating pregnant images, concentrate first on the nude image and then a revealing 1 image. That creates coverage for the entire spectrum. After that a level 3 (underwear) and level 6 (topless) would be the next more appropriate ones. But in general, values of 1/3/9 should be the enough.

### **3. Localization**

Localization refers to the system that keeps the text that's shown to the user in separate files to allow displaying the content in different languages. Localization adds a bit of complexity to the development of the scenes and introduces the potential for bugs as the localization strings are easy to mistype and hard to spot check for errors. And the benefit of multiple language support is a bit of a pipe dream. I can't and even if I could I'm not going to write text for other languages and I doubt anyone else is. In reality the main benefit is to separate the content from the code and make it easier to rewrite the content when necessary.

The way localization works with scenes is that the code when it wants to display some text, it will use a string that we refer to as a localization key. The system then looks for the localization key in a set of files and displays the text associated with that key in the current language.

In addition, the localization will replace certain strings with contextual content. This way the name of a character can be included in the text. Similarly, the proper pronoun to use with a character will be used depending on the gender of the character. A number of more advanced replacements are available and discussed in the following subsections.

String files can be loaded from the standard location as well from mod locations. Unlike with Scene modules, the localization files will replace individual entries. So one does not need to replace the entire contents of a file just to change a few strings.

Localization keys are used in two separate places. First is in the decision fields. The title, body and choice fields of a decision accept a localization key. The second is the `&localize()` and `&localize_1st` built in functions. These functions will also generate localized text from a key and some parameters.

In the event that a localization key is not used, the system will display the key. This can be a short cut for testing things or for emergencies but should not be abused.

## **Localization Voice**

The localization system has the concept of the “speaker”. This refers to who is talking. This is used to determine how to refer to other characters. It will use a term that's appropriate for this relationship. In decisions, the speaker for the title and body will be the narrator. This is not a person that has any relationship with anyone and the terms used will be neutral. Typically the narrator will refer to the MC in the second person, as if it was talking to the player. (This is not automatic, one needs to write the body lines this way).

The choice statements on the other hand use the person making the decision as the speaker. So they should be in the first person. The reason not to hard-code entries that apply to the MC, (like I and you) is that using the replacement codes also paints the names and pronouns in a specific color that can be useful to determine who they refer to more easily.

The system does not support the proper handling of such names in quotes. I.e. if the narrator is quoting what another character is saying one will need to write those quotes in the proper perspective. It will not be applied automatically. The best way to do this is to first create the quote using the `&localize` built-in function and then embed the resulting string as a quote in the narrator's text.

For the `&localize()` built-in function, that one is always in the 3<sup>rd</sup> person. But the `&localize_1st()` function is also available and that would use the MC as the speaker. This all should be expanded in the future to provide more flexibility.

## **Variable Replacements**

There are two types of replacement strings. Variable replacements replace a short string in the localized string with a value of a variable at the time of the replacement. This is indicated in the localized string with the `%XA` syntax. The X stands for an optional formatting character while the A stands for a variable index. At this time we only support one variable per string, so A will always be 1. The X can be one of I or D at this time. More formats can be added at a later time.

When such a string is found, it will replace the `%XA` string with the value of the given variable. When this is used in a **decision title** or **body** (or **bodyif**) statement, it looks like:

**body** “my.loc.key”~\$a

If “my\_loc\_key” maps to “hello there, my %1”, and \$a contains the string “friend”, this will result in “hello there, my friend”. Note that this does not work on **choice** or **choiceif** statements!

In a `&localize()` built-in function call, it can be done like this:

```
&localize(“my.loc.key”, $a);
```

When the code is `%I1`, the value is displayed as an integer (i.e. 4.642 will appear as “5”). When the code is `%D1`, the value is displayed with 2-decimal places (i.e. “4.64”).

If you want to include a “%” sign in the output, you need to specify it as a pair, “%%”.

## ***Names and Pronouns***

To refer to people in the displayed text, one needs to use replacement codes in the localization files. Replacement codes are always enclosed in braces {} and have the form {N:C} where N is the character position in the decision statement and C is the command to tell the system how to do the replacement. There's a maximum of 6 characters that can be replaced: the protagonist (which is 0), the main target (1), the secondary target (2) and the three additional ones (3-5). These match the character positional arguments to the decision statement.

To properly refer to a character, we need some context, and command C provides that context. The context includes:

- sentence structure: is it the subject, object or possessive (he/him/his).
- tone (formal/informal)
- who is the speaker
- do we need an introduction

So the C part of the replacement pattern will look like “xY;Z” where only Y is required. The rest are optional.

- x are the modifier flags. One or more can be specified. Modifier flags are:
  - f: use formal name
  - o: use object
  - s: use possessive
  - u: use object possessive (mine/yours/his/hers). Sorry this is not mnemonic at all.
  - v: use him/herself
  - c: capitalize first (usually used when it appears as the first word of a sentence).
- Z indicates the speaker. If not present, the speaker is figured out from context. The value for the speaker is a number 0-6 which refers to a person in the decision, -1 which stands for the narrator (who uses the second person when referring to the player) or -2 which is used for completely neutral interactions. This is not widely used and probably has not been properly debugged.
- Y is the format to use. This is required. Y can contain one or more characters, terminated with a semi-colon or end of string. Each individual character is replaced as follows:
  - N: first name. If informal, uses nickname if one exists. If informal and is family it will use familiar name (like 'mom' or “aunt Jenny”). Returns “Stranger” if not known. This is the most common format to use when you want to refer to a person by their name.
  - n: first name initial. If informal is uses nickname's first initial if one exists.
  - L: last name. Returns blank if not known.
  - l: last name's first initial.
  - F: first name. Uses nickname if one exists and not formal. Returns stranger if not known. Does not add relationship info like N.
  - A: full name. If informal, uses nickname if one exists (“Johnny Doe”). Shows name even if

not known.

- B: full name, showing nickname in addition to first name (“John (Johnny) Doe”). Shows name even if not known.
- W: full name, if known, otherwise Unknown. Uses nickname if informal. It's the same as A if the target is known.
- w: first name and last initial, if known (John D.), otherwise Unknown. Uses nickname if informal.
- S: short name, nickname + last initial (Johnny D.), shows name even if not known.
- C: internal character id (use for debugging purposes)
- T: title (“Mr.” or “Ms.”),
- I: intro (“my girlfriend”). Note that I differs from the N auto relations in that it doesn't apply to family. It should only be used on the first reference of this character in a dialog.
- P: pronoun, use a pronoun instead of the name. So He/She/Him/Her/His/etc. will be used depending on the flag used.
- p: pronoun like P but in lower case.
- R: name of respect, used to refer to unknown people with a title. This is mostly used for special characters that don't have relationships.
- X: last name, nickname, (e.g “Doe, Johnny”). Used for sorting alphabetically. Name used even if not known.
- Spaces, commas and other punctuation is passed as is, so “L, N” is equivalent to X (assuming it's known)

Note that pronouns will always be shown even if the speaker or listener do not know the character. They would presumably know their gender.

In addition, spaces, periods and commas are passed thru. So again “f. L” maps to “J. Doe”.

When the game starts it will warn you if it finds some simple pronouns, like 'he', 'she', 'his', etc. in the localization files. If you wish to hard code one of these pronouns and don't want to be bothered by the warnings, you can precede them with a caret, like “^he”. All this does is suppress the warning. When the message is displayed the caret will not appear.

## **Flags**

Because of the limitation of 1 variable replacement per string and the fact that it doesn't work on choice statements, we have another option to replace values in a string at run time. It's using flag replacement. Flag replacement is introduced with a question mark, like this: {1:?my\_flag}. This will display the value of the flag “my\_flag” for character 1. You can put an arbitrary value in a flag and have it appear in the display like that. In general have these flags expire after one day to avoid cluttering things.

There's no formatting involved, so it needs to be stored in the proper format first.



## Short Codes

Short codes are a quick way to have the system apply some context conversion and/or use a random string among a list of options. Short codes are introduced with a bang (!) and look like “{1:!g\_person}”. In this case the key is “g\_person”. If the key is of the form “x\_something”, the “x” is a one-character prefix and that indicates a particular conversion. Currently there are two prefixes: “g” and “d”. The “g” prefix stands for the gender of the character. So in the “{1:!g\_person}” example, we take the gender of 1 and replace the “g” with it, getting either “Male\_person” or “Female\_person”. Then we look for the short code for that string.

The second prefix is “d” and that stands for time period. This is used to plug in the current time period into a string. For example, “{0:!d\_day\_night}” is replaced by “day” or “night” depending on the current time of day. The prefix itself is changed to the current tick, so if we are at night, it become “0\_day\_night”. In this case the character is irrelevant, but one still needs to include one.

If there's no prefix, the entire part after the ! is treated as the key. Then when it has a key, it looks for the key in the list of short string mappings. Short string mappings are of the form:

<anything>.ss.key.index

So using the first example with Male\_person. It will find a key in the system with the label “gen.ss.Male\_person.1” which will map to “man”. For another example we have “{1:!g\_insult\_a}”. This will map to a key “Male\_insult\_a” which has a number of options defined:

- gen.ss.Male\_insult\_a.1 an asshole
- gen.ss.Male\_insult\_a.2 a jerk
- gen.ss.Male\_insult\_a.3 an idiot
- gen.ss.Male\_insult\_a.4 a bastard

The system will pick one of those at random.

Note that defining a localization key with “.ss.” in the name automatically makes what follows the “.ss.” a short code. Don't define those unless that's what you want to do. Also, despite the name 'short' it doesn't restrict this to actually short strings. Strings of any length may be defined this way. However, short strings may not contain additional replacements. Anything inside braces {} will be displayed as is.

## 4. Platform API

The Platform API refers to the way the TFEL code connects with the main system (“the platform”) and retrieves or changes data. It does this via either built-in fields or built-in functions. At the time of this writing there are 108 built in fields and 196 built in functions so we are not going to describe them here. When the game runs in dev mode, it will create a file called cheatsheet.txt that will contains all these API calls with a short description. This file is created in the SCW Data/runtime directory.

Built-in fields are also known as object properties. Its a way to get or set a property of an object like a Person, Club, Room, etc. Properties are accessed via the '.' operator, just like in most OO languages. Some properties are read-write, which means that the value can be changed but most are read-only.

For example, if you want to get the name of a person, and that person is stored in the variable @A, you would do this:

```
$name = @A.name;
```

while to set a value you would:

```
@A.pose = "nude";
```

Built-in functions are functions that take arguments and optionally return a value. They work the same way as user-defined functions discussed in the TFL section.

For example, if you want to get A's willpower in the -1 to 1 range you would do:

```
$willpower = &get_attr(@A, "will", 1);
```

or you can do this to make two characters become friends:

```
&make_relation(@A, @B, "friend");
```

All built-in functions must be preceded by a '&' just like TFL functions. It is possible to overwrite a built-in function by using the same name as the built-in. You shouldn't do that unless for very specific situations.

Performance wise, built-in fields are the fastest, then built-in functions and finally user-defined functions.

The cheatsheet.txt file discussed above also contains a lot of additional invaluable information like listing the keys for most objects in the game. This includes attributes, jobs, rooms, relationships, date time formats, laws, relationship modifiers, likes (opinions), AIE flags, etc. Keep it handy!

## Game Objects

To better understand the API one needs to understand the various objects used in the platform. While the platform itself has about a dozen main objects, only seven are exposed via the API. These are the aforementioned Person, Club, Room, Job, Relation, Pregnancy and Gossip. This section will provide additional info on each of these objects.

But before we address the individual objects, we need a better understanding of how objects work. Every object has an id, (known as the "sid" internally, for storage id). These sid's start at 1 and go up by one every time a new object is added. The same universe of ids is used for all objects, so with just an id number we can find the object it belongs to without having to know what type of object it is.

If you have looked at the save files or used the console you probably have noticed that the MC has an id of 3. That's because it's the 3<sup>rd</sup> object created. The first object (the one with sid=1) is the environment (known as World internally) and the second (sid=2) is the Scheduler. The 3<sup>rd</sup> is the the MC. This is not guaranteed to be the case always, but until something big changes the MC's sid will be 3.

If you are using the console you will need to use the sids to reference objects. The console is described further down in this document. In the TFL code, when using objects you normally don't care about the sid. However there are some situations where you must use them. For example, some function calls require you to pass the object as an sid, not the object itself. This is not common, but it happens. A more common usage is for passing containers as function arguments. As mentioned early you can't pass a container to a TFL function or return one. The way around this is to convert the container to a string and vice-versa. But how do you convert a list of objects to a string? Well, you take their sids and convert those to a string.

The API provides a way to access the sid of an object via the id property. For example, A.id returns the sid of Person A. If A is null it will return -1 Note that -1 is the value used for a non-valid or null object. While not used, 0 is valid object sid so make sure to compare with < 0 not == 0 to check for null objects.

Then to convert an sid to an object one can use the `&id_to_char()`, `&id_to_club()` and `&id_to_room()` built in functions. There's currently no corresponding functions for the other object types because that has not been needed yet.

The save files is just a collection of objects. Each line in a save file is the data for a single object. Every line begins the same way, with the object type in the first field (Job, Club, Person, etc.) followed by the sid in the second field. Note that there are a lot of objects in the save file that are not exposed in any way via the API and are only for internal use. The MC object is actually of type MainChar not Person, which internally is a subclass of Person.

## **Person**

The Person object is by far the most complex of all the platform objects. It is what drives the actions of each individual character in the game. (As mentioned earlier the term Person and Character are used interchangeably in this document and throughout the system). A Person object contains a lot of information about a given character, including:

- Name (first name, last name, nickname)
- General information: age, birthdate, home district, gender, sexual orientation, ethnicity
- Attributes, grouped into traits, abilities, skills, vitals and physical characteristics.
- Job
- Game Location
- Opinions about things (called “likes” internally)
- Relations to other characters
- Flags: these are user-defined name/value pairs with an expiration date
- Misc stuff: archetype, action state, cash, debt, image key, current pose, goal

Most of these are accessible via object properties or built-in functions. Some of these require further details:

## **Attributes**

As mentioned earlier attributes are grouped into traits, abilities, skills, vitals and physical characteristics. These are defined via data files (discussed later on). Note however that there's a lot of dependencies between the attributes and the code so adding new attributes may not work as expected. The physical ones in particular are very tied to the image system so adding new ones or changing the existing ones is likely to create problems.

**Traits** are base personality attributes that a character has. These are will power, rationality, compassion, greed and sociability. They are mostly unchanged during the game. While the API provides the ability to change them, the modder should mostly refrain from doing so except in special cases. For instance when creating a new character is fine to set the traits as needed. But allowing a greedy character to become less greedy goes against the spirit of the game and may create inconsistencies.

**Abilities** are similar to traits except that they can change during the game. Again abilities are defined in a data file but they include things like intelligence, lewdness, honor, focus, loyalty, courage, strength and endurance. Abilities behave in a wide range of ways. For example, the focus ability is mostly unchanged during the game. A few things sharpen/dull the focus but these are rare and have limited

power.

Some abilities are closely tied to some traits. For example, the initial intelligence is highly correlated with rationality, while focus is highly tied to will power and honor to compassion. However as the game progresses, this correlation may weaken as the abilities will change independently of their related traits. This correlation is specified in the abilities data file.

**Skills** are used to describe how well a character is at doing a certain activity. These form the basis for how good someone is at their job or at sexual activities. In fact there skills are divided into two types: those that form the basis for jobs (which include things like service, sing and dance) and those that map to sexual activities (kiss, finger, sex, etc.). The initial skills a character gets is based mostly on their traits and abilities. The parameters for this relationship is specified in the skills data file.

Skills, like abilities change during the game. Skills change by reading skill books checked out from the library, by receiving training (for job skills) or by repeatedly doing the action (for sex skills). The last two increase at a rate based on the attributes of the person doing the training or action with.

**Physical Characteristics** are what defines what a character looks like. This includes height, body shape, fitness, hips and butt size, penis/breast size, skin/eye/hair color and hair length. Physical characteristics shouldn't change as they are closely tied to the images. We currently have no way to represent physical changes in the image, so physical changes should be kept to a minimum. It is possible to increase fitness by working out, but the effects are generally small (it is however possible with enough effort to make large changes).

Internally, the physical characteristics are stored as both numbers (in a 0-100 range) and as index into a small number of options. For example, height is stored as a value 0-100 but for mapping it to an image, we turn that into one of three values: short, medium and tall. Those three values are used to then filter the valid images for that character. But for figuring out attraction we use the internal 0-100 number. So while a height of 70 and a height of 95 both refer to tall characters, the 95 one will be taller one and may have a bigger impact to someone who is attracted to tall persons.

In general we use three buckets for each of the physical attributes, small, medium and large. But there are some exceptions. The penis/breast size have an additional huge bucket. The hair length has a bald bucket. Note that the colors are light, medium and dark.

Another complexity worth pointing out is that for images, the body shape and fitness are combined into a single parameter. This leads to the values slim, normal, curvy and fit.

**Vitals.** In version 1.0 some attributes, like happiness, health, arousal, etc. This was a bit problematic for a few reasons so now they have been moved to their own category: vitals. These are attributes that change quite often and reflect the current physical and mental state of a character. These are always in the range of 0-100. Some vitals (health, happiness and fertility) have a base value which is what the character's standard health, happiness or fertility would be and then a current one which reflects their current state.

The API provides three functions to manage attributes: `&get_attr()`, `&set_attr()` and `&update_attr()`. The `&get_attr()` function is used to retrieve the value of a character's attribute in a specific range. Attribute values are stored in the -100 to 100 range (with some exceptions, the physical and the vitals are 0-100). But the `&get_attr()` provides a shortcut to multiply the value to return it in a specific range. This makes it easier to include these in complex formulas. The `&set_attr()` function sets an attribute to a given value and `&update_attr()` changes it by a specified amount. Note that changing attributes should follow the overall conventions for that particular attribute. It's probably worthwhile to search the code to see how much certain actions change a given attribute to come up with a consistent way to update it.

Finally there are a few other loose attributes that don't belong to any category. These are also accessed via the `&get_attr` and `&set_attr` functions but they don't have additional rules around them. Internally these are referred as x-attributes. In version 2.0 some of these x-attributes got stored as their own category, misc attributes. It doesn't change how they are used.

Note that internally there are also job skills. Job skills are not true attributes but they behave as such in some places. They can't be accessed like attributes but have separate API calls to do so. Job skills will be discussed more in the Job object section.

## **Opinions (Likes)**

Persons have opinions about things. For example, a character can like shopping, dislike office jobs and have a neutral opinion of marriage. These are called “likes” internally in the game. Persons also have opinions about other persons. This is the “opinion” value and it's part of the relationship object. So to reduce confusion, we call opinions about things “likes”. Likes have 6 potential values: love, like, neutral, dislike, hate and no opinion. These are mapped to the integer values 2, 1, 0, -1, -2 and 3. In general no opinion behaves like neutral opinion. Characters are normally assigned around 15 opinions when they are created. They can acquire more during game play.

There's a number of built in functions that work with likes. The most common one is `&like_value()` which returns the opinion of a character for a given like. This returns 0 when the Person has no opinion. Meanwhile, `&like_value_exact()` works just like `&like_value` but this returns 3 for no opinion. Normally you want to use `like_value` to get someone's opinion, but in the rare occasions you need to treat no opinion differently from a neutral opinion, then use `like_value_exact`. And `&set_like()` can be used to set someone's opinion.

Likes are defined in the likes.txt data file.

## **Flags**

Flags are values that are tied to a Person, Club or Environment object. For example, when a character reads a book, a flag is added indicating that the character read that book. Similarly, when a character wins an amateur contest, a flag is updated to keep track of that count. Flag values are always stored as strings, the API will automatically convert any numeric values to strings before saving them. Flag names can be anything and this way allows the TFL code to create flags as necessary without having to modify the Platform code. For mods, it is suggested that the mod key is used as a prefix to avoid collisions. Using a generic name for a flag, even if its not used at this time can cause problems if later on the base game code decides to add a flag with that same name.

Flags also have an expiration date. When a flag is set, an expiration is provided which is a number of days. When that number of days has passed, the flag will be removed. This is a way to easily remove temporary variables. If the expiration is set to 0, then the flag has no expiration and will never be automatically removed.

There are two built in functions to deal with person flags: `&get_flag()` and `&set_flag()`. Both require a Person object as well as the flag name. The `set_flag` call requires also the value and the expiration. Passing an expiration of -1 or an empty string for the value results in the flag being removed. The `&get_flag` call returns an empty string if the flag is not set.

Environment flags are also known as global flags. For example, global flags are normally used to track the state of story lines. The `&get_global_flag()` and `&set_global_flag()` built in functions are used to deal with these flags. They operate just like Person flags except they don't need an object argument to be passed in.

And finally, club objects can also have flags assigned to them. This is not used a lot as many club-specific flags are stored in the object of the owner of the club. But when using them, the api functions are `&get_club_flag` and `&set_club_flag`. They operate exactly like Person flags.

## **Game Location**

The Game Location is a sub object of the Person object that indicates location info about a character. The platform object contains the current location, work location and home location for the Person. In the case of a club employee it keeps track of the club id and room id that the person works as well as in some cases (like entertainers that move about), their current work location. The Game Location object is not fully exposed to the TFL code. However, some of the data can be obtained via function calls and properties.

Game Locations come in various types. The most common one are inside clubs. These are identified by a club and room id. There's also home locations, which is where the Person lives. Then there are "named" locations. These are the public locations that can be accessed via the map, like the gym, park, city hall, coffee shop, diner, etc. Finally there's sub-locations, which are not accessible via the map and behave a bit different. These are often referred to as background-only locations. These include places like the patio/bedroom in homes or the alley.

The most important is the `&go_to_loc()` function which is used to move a character to a particular location. This can be named location (like "park" or "gym"), a generic location (like "default", "home", "work" or "away"). To move a person to where they should be at this time use "default". To move a person to somewhere where they can't interact with anyone else, move them to "away". Note that moving characters costs time and energy. There are some exceptions which the system handles. For instance, moving a person home is usually free. A sometimes easier way to move a character is to use the `&join()` function. This moves a person to the location of another person. This is often used to bring people together. And `&move_to_room()` can be used to move a person to a particular room in a club. The `&go_to_loc()` can not be used for this purpose.

To find out where a person is, one can use the `loc_id`, `loc_type` and `loc_str` properties of a Person object. The `loc_id` property is a number value that is guaranteed to be unique for each location and its sole purpose is to determine if two characters are in the same location. The `loc_type` and `loc_str` can be the same, but that does not mean they are in the same place. Finally there are several properties that can be used to test what type of location a Person is at, for example, `is_outdoors`, `is_at_private_loc` and `is_at_named_loc`.

To access background-only locations one would use the `&set_bg()` function. This actually just changes the background without changing the actual location. Using this is a bit tricky and the TFL code must keep track of the location to avoid issues.

## **Job**

The Job object keeps track of the details about where a person works. Unemployed persons have no job object. The data in the job object consists of:

- The job type (it's a string key from the jobs.txt file)
- The salary
- The club id where the job is based (if it's a club job)
- When did the job start

- The number of hours worked in the current time period.

Not all these fields may be exposed by the API. The most important role of the Job object is to tell if a person is employed in a club or not. There's a shortcut property 'club' which can be used to return the club object a Person works for (or null if they don't have a job or don't work in a club).

Salaries are kept as annualized values for hourly employees. This allows for more consistent handling. But these values should be converted to hourly amounts for display purposes.

Persons have job skills, which is a value 0-100 indicating how good they are at a particular job. This considers not only the actual traits/abilities/skills that affect that job (as given by the jobs.txt file) but also the person's experience. The longer they hold a given job, the better they will get at it. Training increases their job skill much faster than the base experience however. Despite the use of the word "skill" here, these are not to be confused with the person's skill attributes. The API does not provide with a way to directly change a job skill. One must change their underlying attributes or experience to do so.

Jobs are grouped into categories. There are around 12 categories and they include service (bartender, waitress, hostess, etc), manual (janitor, driver), craft (cook), clerical (marketing manager), entertainment (singer, dancer) and sex (stripper, hooker). Categories are used to group similar jobs and simplify certain things. For example, for jobs that require college degrees, the category determines the degree. Similarly, how much a person likes a job is based on the category. Each category has a specific like that controls this.

There are also a lot of jobs that have no place in a club and are mostly to create flavor. For example, persons that work as pilots, journalists, accountants, veterinarians, etc. may show up. You will likely not be able to hire these people but you can interact in some other ways. Other jobs are more common, like factory workers, maids, teachers, physical trainers and may be willing to be hired by a club.

## **Relation**

The Relation object is probably the most complex of all the objects exposed by the API. A relation object maintains information between two persons. In a relation, one person is the subject and the other the target. The subject is sometimes referred to as the actor or the self of the relation. The relation objects are always part of a Person object. The Person object of a Relation object will be the subject of the Relation. Note that there will usually be a corresponding mirror relation in the target's Person. So assuming that A and B are persons, it should look like this:

Person A: Relation A:B (subject A, target B)

Person B: Relation B:A (subject B, target A)

When using the API to create a relation, they are created both ways always.

A Relation object will contain the following information:

- The subject and target person ids
- The types of this relation. There are 5 categories of relation types: family, romantic, work, friend and other. Each relation can have a type in each category. In addition, there's a main type which will be the most important of the up to 5 category types. A relation will not necessarily have a type for each category. But if you have a sister and the sister is you employee, then the family type will be sibling and the work type will be employee. And because the sibling relation is considered more important than the employee relation, the main relation will be sibling.

- The date they met
- The opinion and attraction of the subject towards the target.
- A list of relation modifiers
- A list of known gossips (gossips that the subject knows about the target)

Relation types are not defined via data files but they are listed in the cheatsheet.txt file.

To create a relation between two characters use the `&make_relation()` function. The `&have_relation()` function can be used to determine if two characters know each other. The `&get_relation()` returns the relation object between two persons.

## Opinion and Relation Modifiers

The Relation object has an opinion value. This is not to be confused with the opinions on things (likes) discussed earlier. This means how much the subject of the relation likes the target. The opinion between any two persons will always start at 0. The value will then change based on opinion modifiers. When a character (the target) does something that affects another character (the subject), the opinion of the subject towards the target will change depending on what the something is and the personality of the subject. This is handled via relation modifiers.

Relation modifiers are defined in the modifiers.txt data file. For example, lets say that A and B are two persons that know each other. And B then gives A a small gift. This is represented in the game by adding the “small\_gift” modifier to the relation object between A and B. This modifier has a base value of 5. Which means that the A's opinion of B would increase by 5. However, it has a couple of impact values (impact values are used in some of the data files to indicate how a person's attributes impact some values). In this case it has a rationality of 0.01 and an attention of 0.02 impact. This means that if A has a rationality of 70, then the value will be increased by  $70 * 0.01 = 7$  to make it 12. It means that the more rational someone is, they more they will appreciate a gift. Attention is a like, so lets say that A dislikes attention. In this case attention would be -1, so  $-1 * 50 * 0.02 = -10$ , so the overall value will be 2. (Because the like values are in the -2 to 2 range, they are multiplied by 50, also this uses the standard values, so no opinion is treated as 0, not 3).

Relation modifiers also decay over time. In the above example, the small\_gift modifier has a decay of 4, which means that it loses 4 points each year. So after a year has gone by, it will have lost 4 out of the base 5 points, reducing it to 20% of the original. So applying that to the 2 actual value leaves us with 0.4. Some modifiers are permanent (like *close\_family*), some last a long time (*fired\_me*), and some decay quite quickly (*agree*).

The other thing to keep in mind is that the modifiers provide diminishing returns as they go up. So if A has an opinion of 50 towards B, and you add a modifier with a value of 5, the resulting value will not be 55, but rather something like 53.5. It's essentially a logarithmic formula that makes reaching 100 very difficult (although not impossible). It just means that once you get the relation high enough there may not be much value is trying to keep making it go up.

To add a relation modifier use the `&modify_relation()` function. The `&has_rel_mod()` function can be used to determine if a relation has a specific modifier.

The Relation object also has an attraction value. This indicates how much the subject is attracted to the target. Negative values indicate that they are not attracted at all. When a person is not attracted at all towards the gender of the target, the attraction will be set to -100. Forbidden relationships (like due to incest) will also be set to -100. There are two special x-attributes, *m\_attr* and *f\_attr* that define how



much a person is attracted to males and females respectively. The value is 100 to indicate fully attracted and -100 for completely un-attracted. This is used for handling bi-sexuals and generally a persons attraction is multiplied by the appropriate *m\_attr* or *f\_attr* and divided by 100 to get the actual attraction.

For attraction value calculation, there are no modifiers, but it does take into account the subject's likes regarding appearance. So if A likes tall people and B is tall, A is will be slightly more attracted to B.

## ***Knowledge and Fog of War***

Many games have a concept of Fog of War, which is used to hide from the player information they have not been made aware of yet. In this game, Fog of War takes the form of knowledge. While internally knowledge is not actually in the relation object, it is very closely tied to it so will be described here.

The Knowledge objects is what determines what information about person B does person A knows. In general, a persons attributes and opinions (likes) will be hidden from others. As the game progresses, person A may learn some of these values about B. There are many different ways that the game provides for them to learn this information.

The logic as to what is hidden and what isn't is based on what makes for good game play, not what makes sense. Opinion and attraction aren't hidden because the game would be a lot less fun if those were hidden. There needs to be some sort of feedback loop to tell the player what actions were good and which weren't and hiding those results (or showing them in fleeting dialog message) did not lead to a fun experience. Opinions (likes) are always hidden.

Attributes are a bit more complex. Most attributes have a visibility flag which indicates how each attribute is visible to others. The values for visibility are: none (always hidden), visible (always visible), fog (visibility via a standard game action), priv (private, can only be learned when shared), exam (learned via a test or common action), temp (like fog, but changes frequently), special (visible via special actions). All traits have visibility of "fog". Abilities and skills have their own visibility. Physical are always visible except for penis size which is "special" (and it's learned when someone sees the target naked).

Also note that the knowledge functionality only applies to when the subject is a club owner. For all other NPCs they don't really care about these attributes so it's not a factor. This helps keep the data more manageable and improve performance with no practical game effects.

The API provides ways to get the value of attributes and likes from one person as known by another. One should use these API calls in places where that info would affect how a person will act. For instance when deciding to offer a job to someone, it should get the job skill as known by the subject, not their real job skill.

The functions `&has_knowledge()`, `&get_known_attr()`, `&known_job_skill()` and `&like_known_value()` are used for this purpose.

## ***Gossips***

Gossips, also known as rumors, are pieces of information a person has about another person (or themselves). Certain acts of a person that are noteworthy enough will generate a gossip. All persons that witness this act will learn the gossip as true. These persons may then share this information with their acquaintances, depending on the gossip type and their opinion of the acquaintance. The acquaintance will then learn the gossip as an actual gossip, they will not know for sure if it's true or not.

But they will be free to share it with their own acquaintances.

Gossips have a subject (the person of whom the gossip is about). Most gossips have a target too. This is the object/target of the gossip. For example, in the gossip “dumped”, the subject dumped the target. When a person learns a gossip about someone else, that person relationship with the subject of the gossip may receive a relation modifier, depending on whether the gossip is good or bad.

The data stored in a Gossip object is:

- Gossip type
- Subject of the gossip
- Object of the gossip (target)
- The date it occurred
- Whether it's known as a fact

The gossip type determines how the gossip spreads, the relation modifier that it gets assigned, how long the gossip is known before it's forgotten and how “juicy” it is (how likely are people to talk about it). These are defined in the gossips.txt file.

Let's walk thru an example of how gossips work. Let B be a boss and E an employee. B fires E. This creates a gossip “fired” with subject B and object E. The function that creates a gossip is `&new_gossip_fact()`. This gossip is created for both B and E and they both know it as a fact. In both cases the subject is the same. In some cases there may be a mirror gossip with the subject and object reversed but that's rare. In addition some gossips are symmetrical, which means that there's no difference between the subject and object (for example P and Q went on a date). Now B runs into his/her friend F. B may share the gossip that B fired E with F if B likes F enough. Each gossip type has a minimum opinion required to share a gossip when the gossip is about themselves, when they are the object of the gossip or when the gossip is about someone else. In the case of the “fired” these values are 20, 10, 4. That means that the boss is going to share that he fired E with people he/she has an opinion of at least 20, E meanwhile will share it with people E has at least an opinion of 10.

The function `&learn_gossip()` is used to spread a gossip from one person to another. Once F learns the gossip they can share with anyone they have an opinion of at least 4 because they are not involved, so the threshold is a lot less. This doesn't automatically mean they will share every gossip, they will first need to want to interact with someone and will want to share gossips instead of doing something else (their “gossip” like plays a big factor here, the less they like it the less likely they are to spread gossips). Once F learns the gossip, they get a copy of the gossip object that B had, but with the known as fact being false. And because the “fired” gossip has a “`is_tough_boss`” modifier, that means that the relationship between F and B gets a “`is_tough_boss`” modifier.

## **Club**

The Club object contains the information for a club. This includes:

- Base information: name, district, owner,
- State information: expansion level, is operating, maintenance status, damage
- List of rooms
- Posted jobs and jobs available
- Inventory (booze and food)

- Policy values: opening hours, list of prices for products and services, etc.
- Metrics: fame, satisfaction, score, customer cap, revenue share, cash flow, etc.
- Values used by the AI to determine how to manage the club
- Values used by the simulation to determine demand.

A lot of these values are not exposed via the API. In particular, the club AI and the simulation logic are fully implemented on the platform code so none of those values are shared via the API.

Clubs are fairly straight-forward so there's not much more to say about them. But here are some important details.

Some club data is actually stored in their owners Person object as it's impossible to separate the two. The most obvious one is cash. The club's cash and debt are the same as the owner's cash and debt. This is not how real world businesses work, but it's a simplification made to avoid having the MC (and other owners) constantly transfer money back and forth to their clubs.

There's no employee list in a Club object. The employees are actually tied to the room they work in. This however is hid from the TFL code via the `&club_employees()` function. Note that the employee data is also stored in the boss/employee Relation objects and the in the Person's Job object club id. One should be careful when adding or removing employees as its easy to get the data out of sync. It is strongly recommended one look at how the current TFL code does it and either use whatever functions are already in the base game or carefully replicate them. Trying to avoid inconsistent data has been a major priority in the development of this game, and has been avoided for the most part. But this is one situation where it can easily happen.

## **Room**

Of all the objects, Room is the simplest one. The Room object has the following data:

- Type. this drives a lot of the room parameters, as defined in the rooms.txt data file.
- Spot (where in the club map is the room)
- Club id it belongs to
- Number of upgrades
- Construction status
- Workers assigned to this room
- Workers currently working here (some workers, like entertainers and managers move about the club and may be working on rooms other than their assigned room)

The last bullet item is the only complexity with rooms. For example, a stripper is assigned to a dressing room as her work room. However, she moves to the Seating Area, VIP, Private Room, etc. for work. The room one is assigned to is what drives whether it's possible to hire an employee. Their current location is what drives whether they generate revenue for the club and what actions they can take.

## **5. Data Files**

The data files are located in the data folder of the main game. Not all of these files can be included in mod and those that can, not everything is data-driven. Changing some values/fields in the data files

may be ignored by the main program or may cause instabilities and game corruption. It is not feasible to list what can be changed and what can not.

For example, if you add a job via mod *jobs.txt* file (or even adding it directly to the base game's jobs file), random characters may be given that job. But if the job is meant to be a club job, there's no process for an AI club to hire that particular job. Unless you add such logic via a mod.

A more extreme example is that while the rooms in a club are defined in the *room\_types.txt* file, and you can change a lot of parameters there, adding a new room will not work. There logic for the AI to add rooms, as well as the logic that figures out the impact of a room in the club's economy is all in the platform code and it's fairly room-specific. Adding a room to the *room\_types.txt* file will result in a room the MC can add, but will add minimal value to the club and will cause some issues and inconsistencies.

To summarize some of the data below, most data files can be overridden or added to via mods. You just have to include a file with the proper name in the *data* sub-directory of the mod. However, the attributes files (*traits.txt*, *abilities.txt* and *skills.txt*) may not be. These files will be ignored if present in a mod. The physical attributes are not even defined in a file so those also can not be changed via mods. The reason for this is that there's too much internal platform code that depends on the attributes.

If adding a new entry for a given data file that was already defined in the base game, the mod's version will override the base game entry. If there are more than one mod that does this, whichever is loaded last will be the one that's used. So keep this in mind when specifying which mods to load. (The order in the *mods.txt* file matters).

The individual contents of the data files will be discussed here in varying degrees of detail. Each data file has a format description at the top that explains the file contents. Rather below appears a short blurb on what each file is, what information it provides and how it may be safely modified.

## **Abilities**

The *abilities.txt* file defines the abilities, which is one of the four main attribute types characters have. Abilities are things like honor, focus, lewdness, etc. These are like traits but change during the game. The most interesting part about abilities is perhaps it's visibility field. This indicates what visibility does someone has on another's character ability. For example, the intelligence ability has a visibility of "exam" which means that it's hidden but one can find out what it is by testing the other character. This makes this ability an option for a test during the interview process. Meanwhile, honor has a visibility of "fog" which means it behaves like a standard "fog of war" field, in that it's hidden until revealed via some method.

Abilities have a range of -100 to 100. Most abilities have an initial value that's calculated via formula using the trait values as inputs. For example, a character's initial intelligence is 50% rationality and 10% will power. The other 40% is random.

There's no way to override or add new abilities via a mod. Including an *abilities.txt* file in a mod will be ignored.

## **Affinities**

The affinities file is a new file for version 2 which expedites the process to determine what a person cares about. It improves performance by caching a number of factors that then are useful in making a decision. An affinity definition is a list of parameters, attributes, likes and goals indicating how those values affect the affinity. The sum of the absolute value of the weights should be around 100.

For example, the affinity "outdoors" is used to determine how much a person likes being outdoors. It's a factor in deciding where a person will go in their free time. Those that have a high outdoors affinity are more likely to spend their free time at the park or the beach than in the coffee shop or the library.

Affinities are cached temporarily but they are reset whenever a person's attribute changes (except vitals, which change all the time).

## **Archetypes**

The *archetypes.txt* file is used to define common, well-known character classes that create consistent characters across a number of attributes. The archetype file defines some typical parameters for this character. This includes age range, gender probability, sex orientation probability, attractiveness, attributes, opinions and job types. There's still a lot of randomness involved, so that there will be significant differences between characters with the same archetype. But in general they should fit into a similar bucket. (Of course the RNG may not cooperate at all and you may get something somewhat inconsistent, but that should be rare, and it also mirrors real life).

Characters are created as either for a particular game purpose or as pool characters. Pool characters can have an archetype but not always. Purpose characters always have an archetype. This allows us to create characters that fit a particular role in the story. For example, the "princess" archetype models the spoiled, entitled daughter of rich parents. This defines someone who will likely be female, young, straight, irrational, greedy and not compassionate, who likes parties, luxury clothing, shopping, complaining and hates working, sports, reading.

When creating a character, one will normally pass a archetype key to something that fits the story. For example, when creating the initial 7 city commissioners, they are created with the archetype "politician". And when creating a college student, we use the "college" archetype.

The archetypes file can be overridden by mods. However, deleting existing archetypes from the file will cause issues if those archetypes are referenced in the code (platform or TFL). However, updating the settings or adding new ones is completely supported.

## **Calendar**

The *calendar.txt* file is used as a short cut to populate entries into the MC's calendar display. There's a built in function (&add\_to\_calendar) that can do this, but functions defined in this file will be automatically added when the **sched** command is used.

New entries may be added to this file without any issues. And the only impact with removing entries would be some entries not appearing on the calendar display.

## **Gossips**

The *gossips.txt* file defines all the possible gossips in the game. The contents of the file should be mostly straight-forward. One thing to be careful is that the higher the "shareable" values are, the slower the gossip will spread. It is based on the closeness of the person that knows the gossip with the person potentially sharing it with. Closeness is a way to combine opinion and relation type. It's the opinion multiplied by the relation type, where a very close relation (like spouse is high as in 0.99 and a basic acquaintance is 0.1. These values are listed in the cheatsheet. In general 5 is the threshold for better than an acquaintance. For example, the "stole" gossips has share values of 90, 20 and 4. That means that if the gossip is about you, you will share with just only extremely close relations (90, which basically will be your spouse). If you were the target of the gossip (i.e. the person that was stolen

from), then you will share it with people you know well but not everyone (20). And if you are just a bystander you will share with those you know a bit (4). Normally a low number like 4/5 is used for the last field unless the gossip is really good.

Gossips also need two corresponding localization file entries (a short one and a normal one). In the gossips localization strings you can use {1:} and {2:} to refer to the subject and object of the gossip, respectively. Some gossips also create a relation modifier when they are learned. These typically have the format is\_<key>. For example, the “pervert” gossip adds the modifier “is\_pervert”. When doing this you also need to create the modifier as well.

The *gossips.txt* file has no restrictions on what can be changed. In general deleting gossips may cause loss of functionality, but the game should continue to work.

## **Jobs**

The *jobs.txt* file defines all the job types that exist in the game. The jobs file defines a number of parameters associated with the job type. This includes gender probability (how likely is the job to be held by male/female), job category, minimum age, salary or hourly wage, yearly salary range, required lewdness to do job, difficulty (how much does job skill matter to their success), qualifications (formula to calculate job skill from character attributes), importance (to help club AI prioritize hires), etc.

A couple of random thoughts on the jobs.txt file:

- A job with a “perm” flag means that the job is permanent, and anyone with that job will not leave the job on their own. They will not search for new jobs, go to interviews or take a job. They can still lose the job via special code. For example, city commissioners have a permanent job. That means you can't hire them away or even interview them. But they can be voted out. At which point they become unemployed and are able to seek other jobs.
- Every job has a category. Categories are also listed in the cheatsheet.txt file. The category is used to determine what jobs a person like. So instead of having a person like being a singer or a dancer, we just use the entertainment\_work like for that (which applies to all entertainment jobs). College degrees are also awarded by category. So your Business Manager needs a degree in “legal” and your marketing manager needs a degree in “clerical”.
- The formula for calculating job skill is a list of attributes and factors. The attribute value is multiple by the corresponding factor and the products are added together to get the base skill for that character in that job. However, a character's job skill also increases with experience, so the amount of experience is also factored in.
- The attributes listed in the skill formula also define the abilities and skills that can be tested during an interview and trained after they are hired. They attribute needs to also have a visibility of “exam”. For example, the dancer job has “attract” as a component, which is the neutral attractiveness of that person. However because that isn't an attribute (or an attribute with visibility exam), you can't test or train it.

The *jobs.txt* file is tricky to change for a mod. Adding new non-club jobs is fine and you will see random characters take on that job automatically. Changing most of the parameters is fine too. However, adding new club jobs will not work well, except for trivial cases (and cases where there's no impact to club revenue).

## **Law Types**

The *law\_types.txt* file defines the various law types and their categories. The law types categories are mostly defined in terms of how likely a commissioner is to vote for laws in that category (also referred to as their affinity towards the law category).

All the law types require platform code to implement them, so adding/deleting entries to/from this file is not going to work. It is possible to change some parameters of course. But be careful with the law relationships. A law that's replaced by another should never be specified as a requirement for another. (That's because when the law is replaced, then the requirement will break). Similarly, a law that replaces another law should block the law it's replacing.

## **Likes**

Likes is a list of things a person may have an opinion on. The value for each like is one of loves, likes, neutral, dislikes or hates. For example, the “tall” like indicates whether a person likes other persons that are tall. Likes are used to define a person's personality and they affect how they act and what they are willing to do. For example, someone that hates flirting is lot less likely to engage in flirting with you than someone that likes it. And someone that likes attention is more likely to post photos on social media.

Likes have impacts, which are attributes that are used to determine how likely is someone of having a certain opinion. For example, for the “healthy\_diet” has impacts of “will:3 rat:3 stre:3”. This means that their willpower is multiplied by 3, their rationality by 3 and their strenght by 3 and these products are added together. To this some randomness is added. The higher the value the more likely they are to love it and lower it is to hate it. (The values of the impacts can be negative). So basically a strong, rational, high-will power person is more likely to like a healthy diet than someone who is weak, irrational or lazy.

Likes can have co-opinions. This means that when they have one opinion, it's likely that they will also have this one. The healthy diet co-ops are *vegan\_diet*, *working\_out* and *yoga*. So people that have a *vegan\_diet* opinion are more likely to have a similar opinion of healthy diet.

Likes also have opposites and blockers. If a person has the opposite like, they are likely to have this like but with the opposite opinion. For example, for “healthy\_diet”, the opposite is “junk\_food”. That means that if they already have an opinion of junk food, their opinion of healthy diet will be the opposite. So if they like junk food they will dislike a healthy diet. A blocker just does that, it blocks a like if they have another. The blockers for healthy diet are smoking and drug use. If either of these is positive, *healthy\_diet* can not be positive and vice-versa. Likes do not need to have co-opinions, opposites or blockers.

Some likes are “addictive”. These are much harder to change than other opinions. This includes things like alcohol, drug use, etc.

Modifying the *likes.dat* file is straight-forward. You can add new likes without issues or change the parameters of existing one. Deleting existing ones is not recommended as that may cause the game to not work in a consistent/expected manner, but it shouldn't break things.

## **Modifiers**

The *modifiers.txt* file defines all the possible relation modifiers in the game. When a modifier is added to a relation, the opinion of the subject of the relation towards the target gains or loses some points. These points decay towards 0 as time goes by. The modifier data defines how many points are applied

initially and how fast they decay. These points can be further modified by the subject's attributes and likes.

For example, the modifier “agree” which is normally added to a relation when the target agrees with the subject on a particular topic is defined like this:

```
mod agree "Agrees With Me" 5 30 3
traits will:-0.02
likes attention:0.02 leadership:0.01 dominant:0.01
```

The initial points is the 4<sup>th</sup> field of the first line (5 in this case). It also removes 0.02 points for every point of will power that the subject has. The likes values are awarded using 50 times the like numeric value (loves=2, likes=1, etc.). So assuming the subject has will power of -100 and loves attention, leadership and dominant, the formula for the initial points to award would be:

$$5 + (-100 * -0.02) + (100 * .02) + (100 * 0.01) + (100 * 0.01) = 11$$

Note that this is an extreme case. In general the values would hover around 5 with a very rare few extending much past 3 points in either direction.

The decay in this case is 30 (the 5<sup>th</sup> field of the first line). It indicates that in one year the value should decay 30 points. The decay is linear and is relative to the base value, so that means that 5 points (1/6 of 30) would take 2 months to decay (1/6 of 1 year). In the example above the 11 points will decay faster, and reach 0 in the same amount of time. In other words, the time to decay to 0 is the same regardless of the initial value. Higher initial absolute values decay faster than lower ones.

Some modifiers have decay of 0 which means they never go away. These are permanent modifiers which are rare and typically indicate a family relationship.

The last field in the first row is the limit. This is how many modifiers of this type can be attached to a relation. Some modifiers are unique, which have a limit of 1. These are again rare. In general most modifiers have a limit of 3-5. When adding a relation modifier and the relation is already at the limit for that modifier, the oldest one is removed and the new one is added.

Adding new entries to this file or modifying existing ones is fully supported. Deleting existing ones is not recommended as it may lead to game inconsistencies but not major issues.

## **Names**

The *names.txt* file contains all the names that characters can have. Its a fairly simple file. There are three types of names: first names, last names and nicknames. The file is divided into three sections, one for each type of name. First names and nick names also are gender specific. A name like “Chris” that can apply to both men and women must be listed twice, once in the male section and again in the female section.

The nicknames mentioned earlier are independent of the actual name and are things like 'Skip' and 'Sunny' which may not be related to their name. However, many names have short versions that are often used as nicknames, so “Elizabeth” will often be shorten to “Liz”, “Beth”, “Lizzie”, etc. These are listed in the name line for “Elizabeth”. Names also have a weight specific for each ethnicity. For example, Elizabeth has an ethnicity field of w=9,b=3,h=3,a=4. This means that for white women the name has a weight of 9 (making it very common), for black women is 3, etc.If the ethnicity is not listed, it 0 which means the name will never be assigned. Like in this case, Arab women will never get that name (Arabs use the 'r' flag, as 'a' is used for Asians).

There are no restrictions on how this file can be modified as it has no dependencies with the code.



## **Physical**

The *physical.txt* file defines the physical attributes. This is just for loading data into the game as a lot of is dependent on the image system and other code. Changing values in it will likely cause problems so that's not recommended.

## **Regions**

The *regions.txt* file defines some parameters that are specific to a region. Note that currently the only region supported is North America. There would need to be platform code added to support other regions. In addition, for many regions there are not enough characters in their dominant ethnicities to make it work.

The main thing the regions file defines is the ethnic distribution (i.e. percent of characters of a given race) and the names for the districts, common locations (like coffee shop, park, restaurant, etc.) and the AI clubs. This is basically the only things that can really be modified in this file.

## **Room Types**

The *room\_types.txt* file is probably the most complex and important of all the data files. (All files are important, but the parameters defined in this file have the biggest impact on the game). It essentially defines the rooms available to be build on a club.

Note that adding a new room to this file will require additional code to allow the owners to build the room as well as to manually assign workers to it and provide value. This code is on the platform code and while it may be possible to include it via TFL, it would be very hard to do so.

The *room\_types.txt* is very well explained in the comments so won't go into too much depth here.

This file does control certain things about rooms. For example, it controls when a room can built. The requirements are:

- The club level required, or the number of expansions a club has. For example, the security office requires one club expansion.
- Is the maximum number of rooms of this type not reached/exceeded.
- Any other rooms that must be present (for example, a seating area needs a kitchen)
- Are the client actions specified in the “clients” field allowed by city laws? For example, the Private Room can't built until the “priv” action is allowed, which is permitted when the `private_rooms_allowed` law is passed.

There's also a check that the maximum number of rooms hasn't been exceeded but this is not driven by this file.

The `job_type` field defines the job that can be assigned to that room. Every room can have only one type of job assigned to it. For example the Game Room has a job of host assigned to it. That field also indicates how many clients can be served by one worker in an hour. For example, the bar has this line:

```
job_type bartender 2 55
```

This means that the bar supports two bartenders working there and that each bartender at standard skill level can serve 55 drinks in an hour. Note that this value is not per time period (time periods are considered to be 2.5 hours long). The skill level of the bartender (as well as their current energy and

happiness) may affect this value and may be higher or lower. In general a bar can serve up to 275 drinks in a period ( $2*55*2.5$ ).

One other complication is performers. Performers are assigned to dressing rooms. That's where they work for the purposes of tracking number of workers. However, when working, they move around to any room that allows performers (via the `total_performers` field). The AI logic attempts to maximize club revenue by spreading the performers accordingly. This also means that in the seating area and the VIP room, the jobs line refer to the waitresses. It doesn't affect the performers in any way.

Managers are somewhat similar to performers. They are assigned to the auxillary office, and they will tend to roam the club looking for things to do. They can work as one of the service jobs (bartender, waitress, hostess) as well as cook, if the room has an available spot for that job at the moment. So if a bartender is taking a day off, a manager can replace them.

## **Sex Acts**

The *sex\_acts.txt* file contains information about the various sex acts. It's used to drive the sex scenes in the game. It offers a hierarchy of actions, as well as provides the variables used for each gender for each act, as well as the modifiers, skills, likes and gossip keys for things related to this sex activity. It also defines the typical energy and arousal changes due to the activity.

There's not much to this file but some key points are included here. Sex acts normally have a giver and a receiver. The symmetry field if it's true it indicates that both participants are equal, like for kissing.

While some fields in this file can be modified, adding entries will not do anything and deleting existing ones will likely cause the game to not work properly.

## **Skills**

The *skills.txt* file defines the skills used in the game. Skills are attributes that indicate how good a character is at some activity. Skills are divided into two categories: work skills and sex skills. Works skills are used to calculate a character's skill at a given job. Sex skills define how good someone is at a particular sex activity.

Skills have visibility like other attributes. Most of them are set to exam which means they can be tested for and trained. All skills are in the -100 to 100 range, with 0 meaning average.

The initial value of skills are calculated via a formula like abilities. There's a formula provided for each skill, which is just a list of attributes with a factor. The value of a character's attribute is multiplied by the factor and these products are added up to get the overall skill value. The factors don't add up to 1, so the rest is random. For example, a character's base cooking skill is 30% rationality, 20% focus and 50% random.

Skill change as a result of training, reading skill books and performing the appropriate activity. In general, only work skills can be trained and only sex skills improve by doing them. However, there are specific events that will update a character's skill.

There's no way to override or add new skills via a mod. Including a *skills.txt* file in a mod will be ignored.

## **Traits**

The last data file is the *traits.txt* file. This file is the simplest of all the attribute files. The only new thing about this file is the affinity field. This indicates what people think of other people with a given

trait value. The affinities are none, same, pos and neg. A “same” affinity means that people with similar values for the trait will have a positive opinion of each other. Will power and sociability have this, so highly motivated productive people will thing highly of other like them as will lazy and unmotivated people thing highly of other lazy, unmotivated people. The “pos” affinity indicates that people will have a positive opinion of those with a positive value for this trait. This applies to compassion. So highly compassionate people will be appreciated by all others, and those that aren't won't be. The “neg” affinity is the reverse where people will have a positive opinion of those that have a negative value for this trait. This applies to greed. Greedy people will not be liked while the generous will be. Finally “none” means that this trait doesn't affect someone else's opinion of them. This is true of rationality.

There's no way to override or add new traits via a mod. Including a *traits.txt* file in a mod will be ignored.

## **Vitals**

Vitals refer to attributes that change frequently and describe a person's well being and status. Vitals are usually in the 0-100 range, with either 50 or 0 as the default value. For example, health and happiness have a default of 50. Values higher is good, values lower is bad. Others, like arousal and looseness are 0 by default and go up from there.

Health and happiness both have two attributes, the standard one that shows the current health and happiness and base value, that refers to what a person's health or happiness will tend towards. Health is also affected by a person's illness and injury values (which are misc attributes).

## **6. Existing TFL Functions**

This section will highlight a few existing TFL functions that can be used for common purposes. It is recommended that one re-use existing functions if possible instead of creating new ones. The list given here is but a short subset of the available functions and it's also recommended that a would be modder spend some time going over the TFL files in the main game distribution to not only become familiar with the existing functions but the various ways to accomplish certain tasks.

The *tfel/ai\_lib* directory contains the most important and useful TFL functions. However, the other directories may have some worthwhile files. But in general, this directory is the first place to look at.

### **Club Employee Management**

Some functions in *tfel/ai\_funcs/club\_emp.tfl*:

- `&club_emp::likes_club`: how much a person likes or would like working at a given club
- `&club_emp::likes_job`: how much a person likes a particular job type
- `&club_emp::likes_curr_job`: how much a person likes their current job
- `&club_emp::likes_pot_job`: how much a person likes another potential job
- `&club_emp::wanted_salary()`: how much a person wants to be paid to do a given job
- `&club_emp::find_club_employee()`: finds a club employee with a particular job that's currently at work.
- `&club_emp::find_club_employee_any()`: finds a club employee with a particular job but without the being at work requirement..

- `&club_emp::is_job_hourly()`: returns true/false if the given job type is an hourly wage job.
- `&club_emp::has_job_avail()`: returns true/false depending on whether the given club has a particular job opening available.

### ***The Engagemenent.tfl file***

The *tfel/ai\_funcs/engagement.tfl* file contains functions about two characters engaging in an activity. (It has nothing to do with engagement to be married).

- `&engagement::relationship_strength()`: returns a value 0-100 indicating how solid a relationship is. The lower the value the more likely someone is to cheat on their partner.
- `&engagement::relationship_commitment()`: returns a value 0-100 indicating how committed someone is to their top romantic relationship.
- `&engagement::sex_act_engage()`: this is the main function used to determine how much a person wants to engage in a particular sex activity with another person. It's used for both the person making the proposal and the person answering it.
- `&engagement::engage_level_nonrom()`: This is like `sex_act_engage()` but for non-sexual or non-romantic activities. Essentially how much does one person wants to do something with another person.
- `&engagement::saw_someone_naked()`: called when someone sees another person naked to set some flags and relation modifiers.
- `&engagement::spied_someone_naked()`: called when someone sees another person naked without the target knowledge to set some flags and relation modifiers.

### ***The Government Functions***

The file *tfel/ai\_funcs/gov\_funcs.tfl* has a number of functions that deal with the city commission. For example:

- `&gov_funcs::is_commish()`: returns true if the passed argument is a city commissioner.
- `&gov_funcs::commish_of()`: returns the commissioner of a given person.
- `&gov_funcs::commish_for_dist()`: returns the commissioner for a given district
- `&gov_funcs::is_law_active()`: checks that the given law type key is currently active.
- `&gov_funcs::add_corruption()`: adds some corruption to a commissioner
- `&gov_funcs::corruption_value()`: returns the amount of corruption a commissioner has (0-100)
- `&gov_funcs::get_commish_law_affinity()`: returns a value -100 to 100 indicating how likely a commisisoner is to vote for a particular law.

### ***The Relationship Functions***

The file *tfel/ai\_funcs/relationships.tfl* contains a number of useful functions for managing the relationship between two characters. Note that the module name is "relationships" with an "s" at the end.

- `&relationships::know_better()`: attempts a person learn something about another. It can be an

opinion or an attribute value (random). Success is inversely proportional to how well the person knows the target.

- `&relationships::op_agree_on()`: do two people agree on a particular topic, returns 3 (totally agree) to -3 (totally disagree)
- `&relationships::learn_like_if_has_it()`: makes a person learn someone else's opinion on a particular topic.
- `&relationships::learn_random_like()`: attempts a person to learn a random opinion someone else has. Success is inversely proportional to how well the person knows the target.
- `&relationships::learn_new_like()`: attempts a person to learn a random opinion someone else has. In this case will try up to a given number of times to find a new opinion.
- `&relationships::learn_random_attribute()`: attempts a person to learn a random attribute about someone else. Success is inversely proportional to how well the person knows the target.
- `&relationships::learn_good_info()`: like `know_better()` but more likely to learn a useful value. Success is inversely proportional to how well the person knows the target.
- `&relationships::is_valid_romantic_rel()`: checks if two people can have a romantic relationship. This is not for checking if they can have sex, which is more lenient. For example a person who is straight but it's 50% bi can have gay sex, but they can't have a gay relationship.
- `&relationships::is_valid_hookup()`: checks if two people can have sex. Note that this doesn't mean that they want to have sex (use the `&engagement::sex_act_engage()` function for that). It just checks some basic attraction and family constraints.
- `&relationships::get_stranger()`: finds (or creates if none is found) a character that's a stranger to two other characters. Can specify the gender, sex orientation and age of the stranger.
- `&relationships::get_common_relation()`: returns one or more persons in the character pool that are known to two characters.
- `&relationships::get_roommate()`: returns a person the passed character lives with (can be a spouse, girlfriend or a roommate).
- `&relationships::get_roommate_or_close()`: returns a roommate of the passed character or if the person lives alone, a close relationship (like a girlfriend or a family member)
- `&relationships::is_close_relation()`: returns true if two characters have a close relationship. This includes romantic relationships, family members, boss/employee, friends and roommates.

## **The Undress File**

The *tfel/ai\_funcs/undress.tfl* file contains some functions to deal with the clothing characters are wearing.

- `&undress::can_undress_priv_loc()`: returns true if the person can undress, assuming they are in a private location
- `&undress::check_undress()`: checks to see if a person can undress to a given pose. I.e if they are already at the specified pose or more undressed than that, it returns false.
- `&undress::can_match_undress()`: checks to see if a person is more dressed than another person. This checks the actual individual pose reveal values, not the pose names.

- `&undress::undress_to()`: returns the next pose to undress to (e.g. if wearing underwear, returns nude for men, topless for women).
- `&undress::undress()`: undresses a person one level, but following the rules for their location and their character's own modesty. Use this for sex activities in people's homes.
- `&undress::lose_clothing()`: undresses a person one level, but following the rules for their character's own modesty. Use this for sex activities. It's a simpler version of `&undress::undress()`.
- `&undress::should_undress()`: returns true if the person should undress for a given sex activity.
- `&undress::smart_undress()`: like `&undress::undress()` but also considers the individual pose releveal values to ensure a proper progression. Use this for strip teases.

There's a lot of duplicate/rendundant code in this file. It will be cleanup at some point.

## **The Util File**

The file *tfel/ai\_funcs/util.tfl* contains some utility functions that didn't belong in any other category.

- `&util::want_gender()`: returns the gender a person is attracted to (or "" if attracted to both)
- `&util::find_char()`: finds an existing character that matches a number of parameters passed in (like gender, sexual orientation, age)
- `&util::find_chars_with_job()`: returns a list of characters with the specified job.
- `&util::new_char_attr_flags()`: returns the flag string to use when creating a character that you want to be attracted to another one.
- `&util::buff_attr()`: used to improve someone's attribute if below a given threshold.
- `&util::nerf_attr()`: used to decrease someone's attribute if above a given threshold.
- `&util::make_good_looking()`: used to improve a person's physical attributes to make them more attractive. Note: this needs to be called before the character's image is assigned.
- `&util::cum_release_arousal()`: called to update the arousal attribute after cumming.
- `&util::get_weekday()`: used for scheduling events, returns the proper number of days to pass to the **sched** command in order that it runs on a weekday.
- `&util::base_agree_prob()`: Returns the base agreement probability when one person is trying to convince someone else to do something. It incorporates the target's own pliability (see `&ai_likes::follows_orders`) as well as the actor's sociability, and the target's opinion of the actor.
- `&util::can_attend()`: used to see if a person is able to attend events (i.e. not tired, not hurt, not away, etc.)
- `&util::update_like()`: used to change someone's opinion in a given directorion. You can use the built-in function `&set_like()` to set it to a specific value, but more commonly you want to nudege the value up or down one step. You use this function to do that.

## **Other Files**

It's also worth pointing out a few other common situations that have functions already written to handle

them. In some cases these can be used as is or can provide a model for a similar situation.

- In the file *tfel/common\_decisions/court.tfl*, there's a the function `&court::sched_court_appearance()` which can be used to start the courtroom scene.
- In the file *tfel/common\_decisions/fight.tfl* is the function `&fight::fight_loop()` which handles a fight between two people.
- In the file *tfel/common\_decisions/haggling.tfl* there's the function called `&haggling::haggle_exchange()` which implements a haggling mechanism between a buyer and a seller.
- The file *tfel/common\_decisions/photos.tfl* has the function `&photos::add_photo()` which is used when a character takes a photo. Similarly the file *tfel/common\_decisions/social.tfl* has the function `&social::post_photo()` which can be used to post a photo to social media. Note that you only need to do one or the other. If you take a photo, then photo can then be posted to social media later on via the `@photos::view_photos()` action.
- The file *tfel/complex\_scenes/ask\_out.tfl* has the functions to setup a date and go on a date.
- The file *tfel/complex\_scenes/making\_out.tfl* has most of the logic for sex scenes. The function `&making_out::sex_activity_loop()` is a the typical starting point for a sex scene.
- The file *tfel/complex\_scenes/romrel.tfl* has the function `&romrel::wants_rom_rel()` which is used to determine if a character wants to have a romantic relationship with another character. This file also contains the various actions and interactions about starting, upgrading and ending romantic relationships.
- The file *tfel/events/monthly\_pulse.tfl* has two event functions: `&monthly_pulse()` and `&weekly_pulse()`. These are called once each month and week respectively and are used to start story lines or to run other events that are supposed to happen at specific times or intervals.
- The file *tfel/loc\_actions/loc\_lib.tfl* has a few useful functions dealing with locations. The function `&my_private_loc()` asks a character for where they should go for privacy, taking into consideration a lot of factors. The function `&get_home_loc()` returns the location key (which sets the background image) for a character's home. Then `&get_random_person_here()` and `&get_third_person_here()` can be used to return a person that's in the same location as someone else.
- The file *tfel/loc\_actions/move\_char.tfl* has the function `&go_to_random_loc()` which is used to move an NPC to a random location based on the time of day and their interests.
- The file *tfel/owner\_actions/emp\_mgmt.tfl* has a number of really useful functions. In particular, `&quick_raise()`, `&quick_quit()` and `&quick_fire()` are used to give a raise to an employee, an employee to quit their job and to fire an employee respectively.
- The file *tfel/owner\_actions/interview.tfl* has all the logic to conduct an interview. But it has the function `&take_job()` which can be called from anywhere to give a club job to a person. Also useful in this file are `&sched_interview()` which schedules an interview between an owner and a person looking to work in their club and `&is_job_available()` which checks to see if a club has any openings for a particular job.
- The file *tfel/owner\_actions/managers.tfl* has a number of functions to manage the managers duties, like `&get_duties()` which returns the duties assigned to a manager, `&set_duties()` which sets the duties of a manager, `&add_duty()` which adds a duty to a manager's duty list,

`&has_duty()` which checks to see if a manager has a given duty in their list.

- The file *tfel/owner\_actions/owner\_lib.tfl* has additional functions dealing with the owner and employees. `&get_employee_at_work()` returns an employee currently working in the club, similar to those in the *club\_emp* module. `&return_owner_to_club()` and `&notify_owner()` both implement some logic to bring back the owner to his/her club to deal with a situation there although there are some key differences between them. `&emp_value()` returns a numeric value of the worth of a club employee to it's boss.
- The file *tfel/simple\_actions/clothing.tfl* has the function `&got_naked_in_public()` that should be called when someone gets naked in public to alert everyone in the area.
- The file *tfel/simple\_actions/friends.tfl* has the function `&move_out()` to be used when two people that live together stop living together (this could be simply for roommates or for boy/girlfriends that live together, or even married couples that get divorced).
- The file *tfel/simple\_actions/indecent.tfl* has the function `&report_indecent()` that can be used when a character reports another for an indecency violation. The function checks to see if the target person is in violation before allowing the report to go thru which involves bringing the policeman into the scene to assess a fine.
- The file *tfel/simple\_actions/texting.tfl* has the function `&send_specific_text()` which can be used to have a character send a text to another character (usually the MC). It's not meant for major events, just to notify the MC that something happened, and allows the MC to respond with a positive, negative or no reaction. A separate function `&send_specific_text_no_resp()` works the same way but doesn't ask for a reaction from the receiver.

Note that this is but just a tiny subset of the functions already written. And it's only provided as a starting point to show where to find some of the most common and useful functions. There's no substitute from opening the existing TFL files and trying to understand how they work.

## 7. More Information

### Modding Considerations

This section contains a list of basic hints that are critical to successfully creating mods. More recommendations will be listed later on.

You should run your mod in dev mode for testing and debugging. You turn it on via the `dev_mode` setting in the `config.dat` file. Turning this on will also create the cheat sheet file that's essential for writing mods. It contains all the reference information that's not in this document.

Modules can be replaced by loading another TFL file that defines a module with the same name as an existing one. Replacing a module replaces all the contents of the module. It is not possible to replace just one function within a module.

When writing code, break complex expressions and operations into multiple lines. The more complex an expression is the more likely it won't work right.

In the TFL code, empty strings and null strings are equivalent. Test for `""` if you want to see if a string is null, and pass `""` as arguments when you want to indicate a null value.

On an AIE callable, any variable you declare in the `init` section is available in the main body. You can't declare it again, but you may use it. However, if you set a variable to a value in the `init` section, there's



no guarantee that it will still be set to that value in the main body. You should always re-initialize those variables if you do reuse them. Related, you should never make changes to permanent objects in the init section (e.g. like adding cash to a character). The init code is called to determine if the AIE callable should run, so changing a value there would happen even if the AIE callable doesn't trigger.

When creating decisions that are AI only, you don't need to include title or body. You do need to include the choices, but the label strings can be whatever (I tend to use “null”). Similarly, if the decision is for the MC only, you don't have to worry about the choice probabilities or impact values.

## ***Using the Console***

The console is an essential tool for developing mods. The console will allow you to:

- Unit test your code (i.e. force to run specific functions)
- Check the results of what the code did.
- See debugging information in real time.

The console is quite simple and doesn't have a lot of commands. The main commands you should be familiar with are listed next.

### **List**

The list command is used to display a list of all the object of a certain type. For example:

list p

returns a list of all persons and

list c

returns a list of all clubs. The 'r' option is available as well to show all rooms.

For persons, a second argument is allowed and if included it will filter the list to those whose name matches the argument. For example:

list p Susan

will display all characters that have “Susan” as their first, last or nick name. The match is for the full word, so it wouldn't match someone whose name is “Susanne”.

### **Dump**

The dump command shows all the info about a given object (can be person, club, room, the global environment or the current sex session). For example,

dump p 123

displays info about character number 123 and

dump c 32

displays info about club 32. For displaying the environment, no id is necessary (dump e). For characters a third parameter is allowed and when included, it will treat it as another character and then display the relationship between these two characters as well. For example:

dump p 123 456

will dump the info about character 123 including the details of 123 relationship with 456.

### **Run**

The run command is used to force the system to execute a given callable. The callable must be fully qualified. You must pass the proper number and types of arguments. For example:

```
run &util::want_gender char:3
```

runs the function &util::want\_gender, passing as an argument character 3. Arguments must be specified as literal values. Scalars are just numbers, strings, true/false. Objects (characters, clubs, rooms, jobs and relations) must be specified with a prefix and their id. So char:3 is character with id 3, job:999 will be job with id 999. A more complex example would be something like:

```
run &my_mod::my_func char:3 club:32 "string_value" 13
```

In this case it will run the function &my\_mod::my\_func passing 4 arguments: character 3, club 32, the string "string\_value" and the number 13. It is important to remember that the arguments to a run function must be typed, i.e. objects must have the proper prefix, strings must be quoted, etc. In the other console commands the type is figured out from the context.

### **Cheat: set\_attr**

There's a number of "cheat" commands which are used to change values. Note that more complex situations you can create temporary functions to change values as well. This command looks like:

```
cheat set_attr 3 greed 50
```

In this example, it will set the value of the greed trait for character 3 to 50. Note that unlike the run command, you do not need the "char" prefix here for the character ids. You also, and this is very important, do not want quotes on the strings or the char: prefix. The attribute can be a trait, ability or skill.

### **Cheat: set\_flag**

The second useful cheat command is the one that allows setting a flag. Flag are additional values associated with a character. They can be added on demand and they may have expiration dates of when they are automatically removed. For example:

```
cheat set_flag 3 has_widget yes 10
```

sets the flag "has\_widget" of character 3 to the value "yes" with an expiration of 10 days. Note that quotes should not be used unless you want the value itself to have quotes in it. If the expiration is set to -1, then the flag is removed then. If the expiration is set to 0, it will never expire.

### **Cheat: add\_mod**

The cheat add\_mod command allows adding a relation modifier to the relation between a character actor and a target. For example:

```
cheat add_mod 3 103 minor_help
```

adds the modifier "minor\_help" (again no quotes on the string for the modifier id), to the relation between the actor 3 and the target 103. This means that 103 helped 3 in some way so now 3 thinks more highly of 103. There's no corresponding del\_mod command.

### **Cheat: set\_like**

The last cheat command worth discussing here is the set\_like. It works just like set\_attr but it's used to set character opinions (likes). It works like this:

```
cheat set_like 3 hot_weather 2
```

It sets character's 3 like value of "hot\_weather" to 2 (loves). The values are -2 (hates), -1 (dislikes), 0

(neutral), 1 (likes) and 2 (loves).

## **Date**

The date command can be used to convert the internal values used to represent dates into a human-readable string. Internally dates are represented via an integer value that counts up the number of ticks (day periods) from some point in the past before the start of the game (at last check it was January 1, 2023). Using the date command like:

```
date 4512
```

will convert 4512 into the string representation of that date: Morning February 3, 2025.

The command can be used to go the other way too. If you specify a date using the month, day and year, it will return the internal representation for that date. This may be useful when testing a function via the console that requires a date as an argument. In this case you must specify the month, day of month and year as integers as arguments, like this:

```
date 2 3 2025
```

returns 4512. Note that dates must be entered in the standard US format: month, then day of the month then year. There's no provision for passing the day period, so you must add that one by hand (morning is +0, mid-day is +1, afternoon +2, etc.). There's no error checking to make sure that the values passed make sense.

## **Reload\_File**

The reload\_file is used to have the system reload a TFL file. This is used when you make changes to a TFL file and want the changes to be applied immediately without having to exit and restart the game. Normally the command is like:

```
reload_file ai_funcs/util.tfl
```

This will reload the base game file located at tfel/ai\_funcs/util.tfl. For mod files, it requires an argument indicating the mod name, so it will look like:

```
reload_file my_mod mod_file.tfl
```

## **Reload\_Loc**

The reload\_loc command reloads all the localization files. This includes all the mods as well, so no need to specify a mod id.

These are the most useful console commands. Additional commands are available and info about these can be obtained by typing “help” in the console.

One last note about the console. You can cut and paste in it, but using the keyboard (use control-C to copy and control-V to paste). You also need to click on the last row of the console to give it focus. But note that clicking in the console the clicks go thru to the whatever button is behind it. So try to click towards the right side of the screen where there's less chance of a collision with an underlying object.

## **The Debug Log File**

The debug log file is the file created by Unity to which game debugs are written to. This file is normally found at C:\users\<YOURNAME>\AppData\LocalLow\Total Fluke Studios\StripClubWars. It is important to check this log frequently while developing mods as it may show issues with your mod. The **debug** command is used to write any string to this file. The **debug** command prints the file and line number where the debug originated so you don't have to worry about including that info. The

platform code writes a lot of info to this file as well (some may say too much). All of this can be invaluable in debugging your mod.

The debug entries are also written to the console in real time, but that's not as helpful as there's no way to search/filter that content. And the buffer only shows the last 512 entries. But the console debug entries is useful when unit testing code (see the console “Run” command in the previous section).

## **The AI Log File**

Another useful tool for creating scenes is the *ai\_log.txt* file. This file is located in the runtime folder under the main game directory. Whenever an NPC character runs an action or interaction, or an event is run, it's logged to this file. Additional important AI decisions are also logged there. It can be used to see the frequency of when these actions/interactions/events are run as well as to help debug issues. Normally the logging is automatic, but the `&log_ai_action()` built in function can be used to add additional entries to the file.

## **Workarounds**

Here's a list of some of the annoyances I mentioned earlier and some ways to work around them.

- To include a newline in the code put the “\n” in a localized string and then load it via the `&localize` built-in function, like this `set $newline = &localize(“newline”);` (The “newline” loc string key already exists for this very purpose).
- Use the built-in `&round` or `&int` functions when you need to make sure you have an integer. The one time you do need that is when using integers as keys in a container.
- The `!` sign is an identifier prefix, the boolean unary negation operator and used in the `!=` comparison operator. To avoid some parser errors, always put a space after the `!` when using it as a boolean unary negation operator, as in `if(! $x)`
- The `-` unary negation operator doesn't work inside expressions. So if you want to multiply `$a` times `-$b`, you can't do `$a * -$b`. You need to do `-$b * $a`.
- If you have a variable and load a number in quotes in it, it may be evaluated as a number in an expression. This could lead to errors. Try not to do this. Where this can come up is when using list containers. List containers have indexes of ordinal numbers as quotes. If you must, have them not be the first term in an expression. So if `$i` can have one of these values, do “something” == `$i` instead of `$i` == “something”. Or use a number instead of “something”. In an expression, the first value sets the type that other values will be evaluated in.
- Note that variables are initialized to the value of “0” by default. This can also cause problems when doing comparisons with these variables. You should always initialize scalar variables before using them.
- Sometimes you need to pass a null object to a function or return a null object to indicate it wasn't found. The right way to do this is to pass a declared variable that has not been initialized. So you want to set a character object `C` to a null value, set it to another character object that has not been assigned yet.
- You can't combine multiple boolean operations in the same expression. So `$a > $b & $b > $c` doesn't work. That's because, that's evaluated left to right so it's `(( $a > $b ) & $b) > $c` which is nonsense. You need to put parentheses to establish precedence: `( $a > $b ) & ( $b > $c )`.

- To use the **foreach** statement on a container that has key/values (i.e. a hash/dictionary), use the `&keys` built in function to get the keys from the container and then do the **foreach** on the return value from `&keys`.
- You can't put a **foreach** loop inside a **while** loop or another **foreach** loop. If you need this, you need to either convert the inner **foreach** loop to a **while** loop or move the code to another function.
- You can't define new constants in the TFL code. However, you can add a constant to the `user_config.dat` file and it will become available as a constant in the TFL code.
- To have a timeout on a callable that triggers only if the code reached a specific line in the code, set the timeout value to 0 early on in the callable code and then manually set the timeout on the affected character(s) at the right time. To do that you set a flag on the character with the name `"_aie_<name of callable>"` and set to expire in the number of days you want the timeout to last. The name\_of\_callable has to be fully qualified, including prefix and module name. This is a bit tricky, so try and find an example on the released code (this is good advice for most entries in this list).
- Sometimes you have variables that are storing numbers as strings. There's a few ways that this can happen, the most common being via the `&get_flag()` built-in function. That function always returns values as strings. If you then compare that variable with another, the comparison may happen as strings, and while "4" and "5" will be the same comparison regardless if it's done as strings or numbers, "4" and "12" won't be. To insure the comparison is done as numbers, do something like: `set $a = 0 + &get_flag(A, "x"); if($a < $b) {...}` . Make sure to always put the 0 as the first term in the addition.
- To schedule an event to run on a specific day of the week `$want_dow`, you need to first get the current day of the week `$curr_dow` using `&curr_day_of_week()`, and then making sure to set the value of days in the **sched** command to `$weeks*7 + $want_dow - $curr_dow`, for some integer value of `$weeks`.
- Functions can't return containers. To return a list of values, concatenate them into a string with a separator via the `&unsplit()` built-in, return that string and then use the `&split()` built-in to convert the string back into a container. It's also not possible to pass containers as arguments to non-built in functions, so the same workaround may be used for those situations. In addition to the `&split()` and `&unsplit()` built in functions, there are the `&cont_to_str` and `&str_to_cont` built in functions that can also be used for hash/dictionary type containers.

## ***Things You Will Do Wrong A Lot***

These are mistakes I'm constantly doing...

- Trying to set a variable in a **var** statement.
- Extra semi-colons are bad. Don't put them when you don't need to. For example, there's never a semi-colon after a closing brace, so decision statements and the **init** block of a callable don't have a semi-colon after.
- Using `&&` and `||` instead of `&` and `|` as binary boolean operators.
- The variable used in a **foreach** loop can not be used in another **foreach** loop in the same callable. It however, may be used outside the loop after the loop. And after the loop ends, the value is the last value executed in the **foreach**, which can be helpful if using **break** to end the

loop.

- The **foreach** loop operates on the values of the container, not the keys.
- Forget the **return** statement on callables that don't return values. Doing so may result in the execution not returning properly and instead executing the code for the next callable in memory. It doesn't happen always, and it's actually somewhat rare but I can't seem to squish this bug.
- Forgetting that the **allow** and **aiprob** fields of an AIE callable preamble require a semi-colon at the end.
- Trying to use the ~ notation in a choice. The ~ notation only works in body and title.
- Assume that the MC object's opinion of its relations are valid. You may prevent things from happening between NPCs if they don't like each other, but you should always give the MC the ability to act regardless of what the internal opinion or attraction values are.
- The keys in choice statements of a decision must string literals. These strings can be numbers, like “3” or “45”, but if so, they all must be numbers. You can't have some strings and some numbers in the same decision. Also, if using a number, it should be an integer. Using floating point numbers will likely fail in the comparisons afterwards. So in general, try not to use numbers.
- Variables can't be keywords. There's some unexpected keywords that one will forget about, like **details**, **choice** or **allow**. These can't be used as identifiers.
- Passing a container in a function or return a container from a function. You must convert them to strings to do so. (This doesn't apply to built-in functions).
- Specifying a function call as a parameter to another function. Call the function and store the result in a variable, then pass the variable.
- Putting an expression or function call as container index. Again, put the result of the expression or function call in a variable and then use the variable as the index.
- Global variables. Every variable is scoped in just their own callable. If you need to save values globally, store them as global flags. But note that this is relatively inefficient so use it only when absolutely necessary.