Quick introduction to Hooks

This guide is intended for people new to PAYDAY 2 Lua modding and explains different methods of hooking functions and when to use them. I'll go over the most commonly used methods of function hooking and explain how they work and why we need them.

Why bother using hooks?

You may ask yourself: Why all this complicated stuff? Can't I just copy and paste the original function and make my edits to it? While yes, you could do a complete *function override*, you really shouldn't unless it is absolutely unavoidable. In the worst case this will crash the game when an update changes some things in that function and you will have to copy and change the new function again to fix it. You will also run into the risk of breaking mods that rely on the same function, as your mod would completely undo changes of other mods if it loaded afterwards.
The major takeaway is therefore: Doing proper function hooks increases compatibility with other mods and game updates and is less likely to break over time. With that out of the way, let's see what kind of hooking techniques exist.

Types of hooks

PostHook

Depending on what your mod is trying to achieve, you will have different needs when hooking functions. The most common scenario is probably adding additional data or code on top of an existing function after it executes. For this we generally use SuperBLT's *PostHook*, which registers a custom function that is executed whenever the game executes the target function. Using a *PostHook* is simple and can be used if you don't need to change the functionality of the original function and just want to add some additional code.

Let's look at an example: We want to change the HP of the regular street cops. We can do this after the cop's stats have been initialized, which happens in the function `CharacterTweakData:_init_cop`. Looking at this function in the game code we can see that the cop's health is set here. Since we just want to change the HP and still want all the other things to be initialized like usual we can add code in the form of a *PostHook*.
We would do that like this:

```
Hooks:PostHook(CharacterTweakData, "_init_cop", "our_unique_hook_id", function (self)
	self.cop.HEALTH_INIT = 10
end)
```

When the game runs the original `CharacterTweakData:_init_cop` function, all the cop data will be initialized as usual, but immediately afterwards, BLT will execute our hook function, changing the cop's health to the value we supplied.

The function you specify for the *PostHook* will be supplied with the same arguments that the original function will be called, so if you need them, you can specify them.
Note that `CharacterTweakData:_init_cop(presets)` is functionally identical to `CharacterTweakData._init_cop(self, presets)` (specifically note `.` instead of `:` between `CharacterTweakData` and `_init_cop`). This is important for your hook function if you plan to make use of the function arguments and the reason why you usually see `self` as the first argument in a hook function even if the original function doesn't have it.
If you are unsure about wether to include `self` in the list of arguments for your hook, just remember that if the function is defined with a `:` in the game code, the first argument to it in a hook should be `self`.

If you need access to what the original function of your hook returned you can use `Hooks:GetReturn()` which will return any values that have been returned by hook functions and the original function call that ran before your hook function. Let's say you made a new custom enemy and for making it work properly you need to add it to the character map (a list of all enemies the game goes over and generates contour mappings for). The character map is returned by the function `CharacterTweakData:character_map` and the function itself creates and returns a local table with all characters in it.

```
Hooks:PostHook(CharacterTweakData, "character_map", "our_other_unique_hook_id", function (self)
	local char_map = Hooks:GetReturn()
	table.insert(char_map.basic.list, "custom_enemy_name")
	return char_map
end)
```

You can see that it easy to access the return value and change it (in this case the return value is a table and we insert an entry into it. Another thing that you should notice is that you can return values from a hook, this will override the return value of the original function (and any function hooks on the same function that came before yours).

PreHook

Very similar to a *PostHook*, the only difference is that it will be executed before the original function is called. Less commonly used but useful if you need to change some values or add additional code right before a function call. Note that setting fields that are created or set in the

function in the original code will have no effect since your code runs before the original function and the original call will just override any values that it sets.

Another niche use case for a *PreHook* is changing function arguments that are of table type, as tables are passed by reference and you can therefore change the content of the table which will then be passed to the original function. As other data types are passed by value this only works for tables.

Function wrapping

The above mentioned hooking methods should cover a lot of usecases already, but there are some cases where they are not usable. Let's say you want to change the arguments the original function is called with or stop the function from being called based on some condition. This simply can not be done with a *PostHook* or *PreHook* since all function arguments except tables are passed by value and *PostHook* or *PreHook* will always run the original function.
In this case, you need to do a *function wrap*, often referred to as *old_init*. This involves saving the original function into a variable and then overriding the function with a new one. In the new function you then call the original function manually and do whatever else you need to do.

```
local build_suppression_original = CopDamage.build_suppression
function CopDamage:build_suppression(amount, ...)
	if amount == "max" then
		amount = 2
	end

	return build_suppression_original(self, amount, ...)
end
```

Some notes on this code:

- Using `...` in the function arguments represents any number of additional unspecified arguments. We can use this to make sure we pass every argument that our function is called with to the original function call without actually caring what they are. If you need some of the actual arguments, you can simply list all arguments up to the ones you need and then follow them by `...`. It is suggested to always use `...` at the end of the argument list, even if you already listed all of the original arguments in case any other mod or game update adds additional arguments to that function call.
- We have to make sure to return whatever the original function is expected to return, as returning a wrong type or nothing at all can lead to crashes that are hard to pin down.

So you chose to override after all

If none of the methods above can be applied to what you want to do, you will have to override the entire function. However, SuperBLT provides a way to do this in a way that at least keeps any hooks made by other mods intact.
An example could be the following:

```
Hooks:OverrideFunction(GroupAIStateBesiege, "assign_enemy_to_group_ai", function (self)
	-- Your code here
end)
```

Note that if another mod used SuperBLTs *PostHook* or *PreHook* on the same function, they will still be called and you can maintain *some* compatibility with other mods. Redefining the function without making use of `Hooks:OverrideFunction` will remove all hooks that were made by mods that ran before yours which could lead to unexpected behavior.

Revision #7
Created 15 January 2023 19:38:07 by Hoppip
Updated 1 March 2023 15:35:42 by Hoppip