# Using Theme Hooks

A better way to modify a child theme via ***functions.php*** is to use hooks. If you have never heard of theme hooks before, think of them as little anchors in a theme’s files that allow you to add content, functions and other stuff right there, without having to edit the core file themselves.

There are two types of hooks: ***action*** hooks and ***filter*** hooks. Action hooks allow you to add custom functionality to existing functions. Filter hooks are a way to modify the functions present in the hook’s location.

# Overriding parent theme functions in your child theme

* Pluggable functions
* Function priority
* Removing functions from the hook they’re attached to

## Pluggable Functions

Pluggable functions in WordPress are functions that can be redefined by plugins or themes. This means that if a theme or plugin defines the same function, it will override the default implementation. This is useful for allowing customization and extending the functionality of WordPress without modifying its core files.

## Why Use Pluggable Functions?

* **Flexibility:** Allows themes and plugins to modify core functionality without changing core files.
* **Customization:** Enhances the ability to customize WordPress behavior to fit specific needs.
* **Maintainability:** Keeps customizations separate from the core, making updates easier and more stable.

## How to Create a Pluggable Function? Step1: Define the Function

## When you define a pluggable function, you need to check if it already exists. This prevents the function from being redefined if it has already been overridden by a plugin or theme.

**Here’s a simple example:**

*if (!function\_exists('my\_custom\_function')) {*

*function my\_custom\_function() {*

*// Your function code here*

*echo 'This is my custom function!';*

*}*

*}*

* **`if (! Function\_exists(‘my\_custom\_function’)) {….} `:** This checks if **`my\_custom\_function`** has already been defined.
* **`function my\_custom\_function () {…...} `:** This defines the function if it doesn’t already exist.

## Step2: Use the Function in Your Code

You can now use **`my\_custom\_function()`** anywhere in your theme or plugin. If a plugin or another theme defines this function, their version will be used instead of the one you provided.

**Example in Practice**

Imagine you have a function that outputs a custom greeting. Here’s how you might set it up as a pluggable function:

**In Your Theme or Plugin File**

*if (! function\_exists('custom\_greeting')) {*

*function custom\_greeting() {*

*echo 'Hello, welcome to my website!';*

*}*

*}*

**Overriding the Function in a Plugin**

Now, let’s say you have a plugin that needs to provide a different greeting. In the plugin file, you would define the same function:

*if (! function\_exists('custom\_greeting')) {*

*function custom\_greeting() {*

*echo 'Hi there! Thanks for visiting!';*

*}*

*}*

**Important Considerations**

* **Naming Conflicts:** Be careful with function names to avoid unintended conflicts.
* **Execution Order:** Ensure the pluggable function is defined early enough to be overridden, typically in the **`function.php`** file of your theme or in a core plugin file.
* **Scope and Context:** Pluggable functions are mainly used for backward compatibility and should not be overused. For more flexible customization, consider using hooks and filers.

## Best Practices

* **Use Hooks and Filters When Possible:** Pluggable functions are a legacy method of customization. Hooks and filters provide more flexibility and are the preferred method in modern WordPress development.
* **Documentation:** Clearly document your pluggable functions so that other developers know they can override them.
* **Testing:** Test your theme or plugin with and without the pluggable function being overridden to ensure it behaves correctly in both scenarios.

By understanding and using pluggable functions appropriately, you can create more flexible and customizable WordPress themes and plugins.

## Function Priority

In WordPress, function priority determines the order in which functions attached to a specific action or filter hook are executed. When you add a function to a hook, you can specify a priority. The lower the number, the earlier the function runs. If no priority is specified, the default is **`10`**.

## Adding Functions to Hooks with Priority

When you attach a function to an action or filter, you can specify the priority as an optional parameter in the **`add\_action`** or **`add\_filter`** function.

**Syntax:**

add\_action('hook\_name', 'function\_name', $priority, $accepted\_args);

add\_filter('hook\_name', 'function\_name', $priority, $accepted\_args);

* **`hook\_name`:** The name of the hook to which you are attaching the function.
* **`function\_name`:** The name of the function you are attaching.
* **`priority` (optional):** The priority at which the function should be executed. Lower numbers correspond to earlier execution. Default is **`10`**.
* **`accepted\_args` (optional):** The number of arguments the function accepts. Default is **`1`**.

## Examples

**Adding a Function with Default Priority**

If you don’t specify a priority, the default value of **`10`** is used:

*add\_action('init', 'my\_custom\_init\_function');*

*function my\_custom\_init\_function() {*

*// Function code here*

*}*

**Adding a Function with a Specific Priority**

You can specify the priority to control the order of execution:

*add\_action('init', 'my\_custom\_init\_function', 5);*

*function my\_custom\_init\_function() {*

*// Function code here*

*}*

In this example, **`my\_custom\_init\_function`** will run before any other function hooked to **`init`** with the default priority of **`10`.**

## How Priority Affects Execution Order

Consider the following example where multiple functions are hooked to **`init`** with different priorities:

*add\_action('init', 'first\_function', 10);*

*add\_action('init', 'second\_function', 5);*

*add\_action('init', 'third\_function', 15);*

*function first\_function() {*

*echo 'First function executed';*

*}*

*function second\_function() {*

*echo 'Second function executed';*

*}*

*function third\_function() {*

*echo 'Third function executed';*

*}*

The output will be:

*Second function executed*

*First function executed*

*Third function executed*

* **`second\_function`** runs first because it has the lowest priority **(`5`).**
* **`first\_function`** runs next because it has the default priority **(`10`).**
* **`third\_function`** runs last because it has the highest priority **(`15`).**

## Best Practices for Setting Priorities

* **Default Priority:** Use the default priority **(`10`)** for most functions unless you need precise control over the execution order.
* **Early Execution:** Use lower priority numbers **(e.g., `1-9`)** for functions that need to run early.
* **Late Execution:** Use higher priority numbers **(e.g., `11-20`)** for functions that depend on other functions having run.
* **Consistency:** Maintain consistent priority usage across your project for readability and maintainability.

## Example Scenario: Modifying Content

Suppose you want to modify post content in steps: first, append some text, and then wrap the content in div. You can achieve these using priorities:

*add\_filter(‘the\_content', 'append\_text\_to\_content', 10);*

*add\_filter(‘the\_content', 'wrap\_content\_in\_div', 20);*

*function append\_text\_to\_content($content) {*

*return $conten t. ' - Appended text';*

*}*

*function wrap\_content\_in\_div($content) {*

*return '<div class="custom-wrapper"> ‘. $content. '</div>';*

*}*

Here:

* **`append\_text\_to\_content`** runs first (priority **`10`**).
* **`wrap\_content\_in\_div`** runs after (priority **`20`**), ensuring the appended text is also wrapped in the div.

## Conclusion

Understanding and managing function priority in WordPress allows you to control the execution order of your functions, ensuring they run in the correct sequence relative to other functions. This is especially important in complex projects where multiple functions interact and depend on each other. By strategically setting priorities, you can achieve the desired behavior and maintain a well-organized codebase.

## Removing Functions from Hooks

To remove a function from a hook, you can use the **`remove\_action`** or **`remove\_filter`** function. These functions work similarly to **`add\_action`** and **`add\_filter`,** but instead of adding, they remove the specified function from the hook.

**Syntax:**

*remove\_action('hook\_name', 'function\_name', $priority); remove\_filter('hook\_name', 'function\_name', $priority);*

* **`hook\_name`:** The name of the hook from which you want to remove the function.
* **`function\_name`:** The name of the function you want to remove.
* **`priority` (optional):** The priority at which the function is hooked. This must match the priority used when the function was added. Default is **`10`**.

**Example**

Let’s walk through an example. Suppose a theme or plugin has added a function to the **`wp\_head`** hook”

*add\_action('wp\_head', 'custom\_header\_function', 10);*

*function custom\_header\_function() {*

*echo '<! -- Custom Header Content -->';*

*}*

If you want to remove this function from the **`wp\_head`** hook, you would do the following:

*remove\_action('wp\_head', 'custom\_header\_function', 10);*

## Important Considerations

 **Priority Matching**: The priority used in **remove\_action** or remove\_filter must match the priority used in add\_action or add\_filter. If the priority is different, the function will not be removed.

 **Order of Execution**: Ensure that remove\_action or remove\_filter is called after the original function has been added to the hook. This often means placing your code in a later hook or ensuring it runs after the initial hook setup.

## Example in a Plugin or Theme

To ensure that the function is removes properly, you might place the **`remove\_action`** call in the **`functions.php`** file of your theme or in a custom plugin:

In **`functions.php`** of Your Theme

*function remove\_custom\_header\_function() {*

*remove\_action('wp\_head', 'custom\_header\_function', 10);*

*}*

*add\_action('after\_setup\_theme', 'remove\_custom\_header\_function');*

In a Custom Plugin

*function my\_custom\_plugin\_remove\_header\_function() {*

*remove\_action('wp\_head', 'custom\_header\_function', 10);*

*}*

*add\_action('init', 'my\_custom\_plugin\_remove\_header\_function');*

**Real-World Scenario**

Imagine a scenario where a plugin adds some scripts to the footer using the **`wp\_footer`** hook, but you want to remove this script for performance reasons:

**Plugin Adds Script**

*add\_action('wp\_footer', 'plugin\_footer\_script', 10);*

*function plugin\_footer\_script() {*

*echo '<script src="https://example.com/script.js"></script>';*

*}*

**Removing the Script**

In your theme’s **`functions.php`** file or a custom plugin:

*function remove\_plugin\_footer\_script() {*

*remove\_action('wp\_footer', 'plugin\_footer\_script', 10);*

*}*

*add\_action('wp\_loaded', 'remove\_plugin\_footer\_script');*

## Conclusion

Removing functions from hooks in WordPress allows you to customize and control the behavior of themes and plugins without modifying their core files. This is particularly useful for maintaining customizations across updates and ensuring that your site behaves as expected.

By understanding how to properly use **`remove\_action`** and **`remove\_filter`**, and considering the order of execution and priority matching, you can effectively manage and customize your WordPress site.