

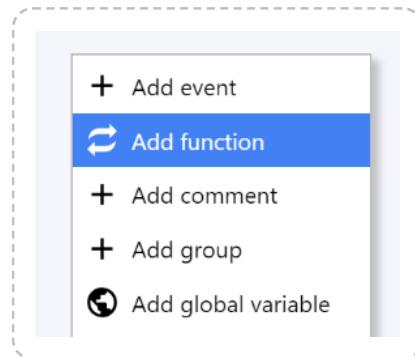
FUNCTIONS

View online: <https://www.construct.net/en/make-games/manuals/construct-3/project-primitives/events/functions>

Functions are special kinds of event blocks that can be called from actions. They are designed to be analogous to functions in real programming languages. Using functions can help you organize event sheets and avoid having to duplicate groups of actions or events.

Adding a function

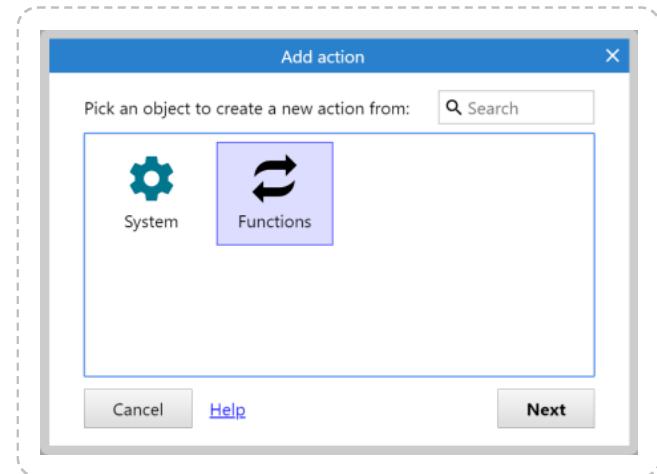
In the event sheet, functions are represented as a different type of event block. To create one, use the *Add function* menu option instead of *Add event*.



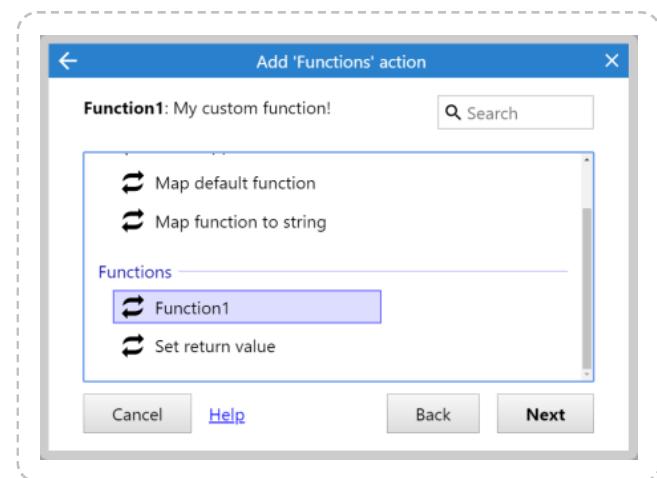
When you select this the [Add function dialog](#) will appear for you to fill in details about the function. Once created, the function appears in the event sheet similar to a normal event, but with a special function icon and *On function* text at the top.



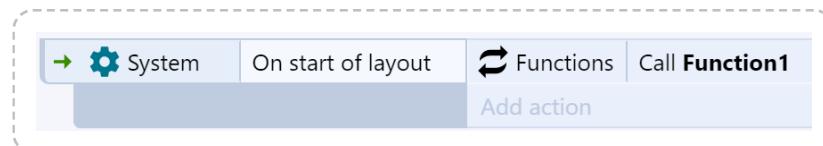
You can add [conditions](#), [actions](#) and [sub-events](#) to functions, just like you can with normal events. However functions do not run unless you call them in an action. Once you've added a function to your project, a new special *Functions* object appears in the [Add action dialog](#), next to the System object.



When you choose this object, it displays the functions in your project as if they are actions.
(There are also some other built-in actions that relate to functions.)



Choosing the function adds an action that calls (runs) the function.

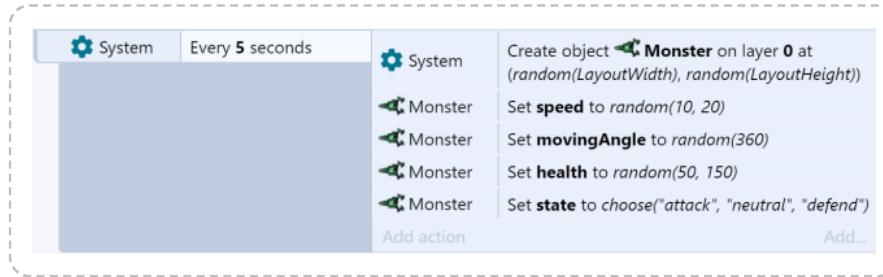


Functions are similar to [custom actions](#), but not associated with a specific object type or family. This action will run the corresponding *On function* event, including testing its conditions, running actions, and running any sub-events, and then return to the original action and continue from where it left off.

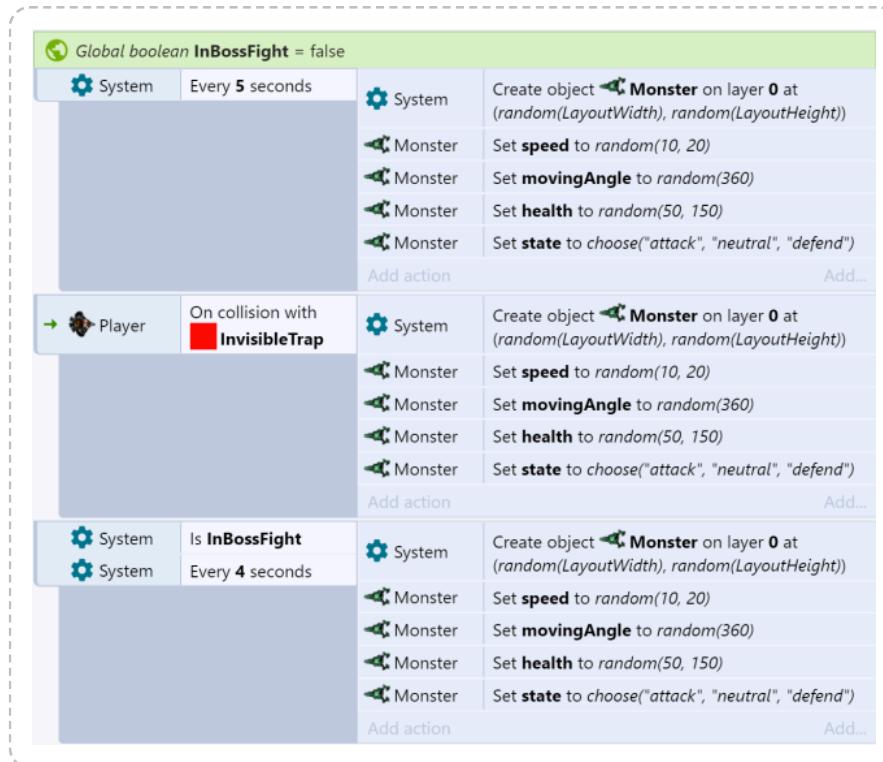
Functions are global. This means you can call a function from anywhere in your event sheets, even if the function is in a different event sheet that is not included in the event sheet you call it from.

Using functions

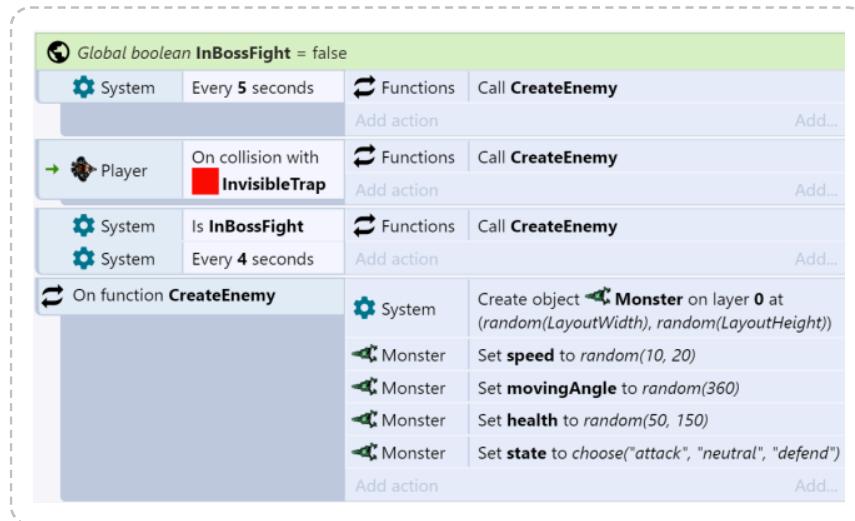
A good example of using functions is to eliminate repeated sets of actions or events. For example suppose you create an enemy with random properties every 5 seconds using this event:



Suppose there are two other events where you want to create an enemy the exact same way: one when a player walks in to a trap, and another one every 4 seconds when in a boss fight. Without functions, you may have to copy-and-paste the same actions multiple times, like this:



Notice this is becoming inconvenient. There may be times you need to repeat the actions in even more places. If you want to make a change, you then have to find every place you repeated the actions, and repeat the change. We can remove the repetition using functions. By creating a *CreateEnemy* function which has the repeated actions, we can replace all the repeated actions with function calls, like this:



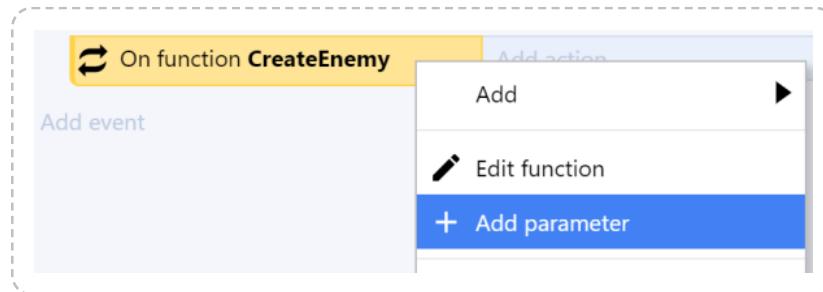
This works identically to the previous events, but is much shorter and more convenient. We can call the *CreateEnemy* function anywhere in our events we want to create an enemy, and it uses the same set of actions in the corresponding *On function* event.

It is often useful to split many parts of your events in to functions like this, so they can be conveniently re-used across event sheets.

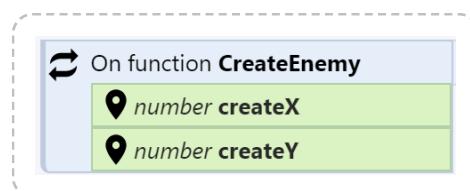
Parameters

When calling a function, you can also pass parameters. These are numbers or strings that are made available to the function. For example, the *CreateEnemy* function from the previous example could be modified to take two parameters: the X and the Y co-ordinates at which to create the enemy. This helps functions to be made more general purpose by using extra information from the action calling the function.

To add a parameter to a function, use the *Add parameter* menu option when right-clicking the function. (Note you need to right-click on the header or margin, since if you right-click a condition, it will show a menu for the condition instead.)

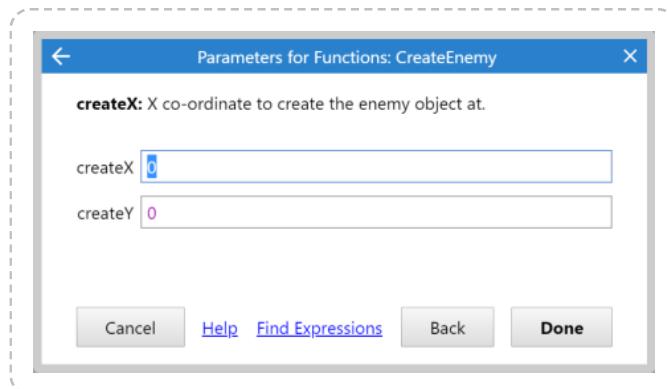


When you select this the **Add function parameter dialog** appears for you to fill in details about the parameter, including its name, description and type. Parameters appear similar to [local variables](#), but inside the function block.



Parameters work very similarly to local variables - you can use them in expressions, compare them, and set them just like any other kind of local variable. Similar to local variables they are limited in scope to just the function event and its sub-events.

Now when you call the function, you can also provide the parameters. Notice the name and description you set for the parameters are used. These appear like parameters for any other action, but they will set the values of the parameters when calling the function.



Returning values from functions

Functions can also return a result. For example, a factorial function could calculate the mathematical result and return it.

By default, functions have a return type of *None*, meaning they don't return any value. This also means they are used as actions. However if you set a return type of *Number*, *String* or *Any*, the function returns a value. This also means it is used as an expression instead, so it won't appear as an action.

A function can set its return value using the *Set return value* action in the built-in Functions object. It can then be called using it as an expression, such as:

```
Functions.MyFunction
```

Parameters can also be added in parentheses, e.g.:

```
Functions.MyFunction(1, 2, 3)
```

The expression returns the value set by the *Set return value* action in the function call.

Functions which return a value will also appear in the [Expressions dictionary](#), also show up in autocomplete, and also show call tips when entering parameters, just like other expressions. In summary, while functions with no return type are essentially custom actions, functions with a return type are essentially custom expressions.

Picking

Normally, calling a function will run the function with picking reset. That means if an event picks some instances with conditions, then calls a function, the function runs with all instances picked again, ignoring the fact that conditions previously picked some instances.

Enabling *Copy picked* on the function changes this so the function keeps the same picked instances when it is called. This can be convenient for making a function that affects a single instance, for example - its actions will run on the instance picked by the caller, rather than having to pick the instance another way (e.g. by its UID).

Note that if the function changes which instances are picked with its own conditions, that does not affect the place that called the function. When returning after the function has finished, any changes the function made to picking are discarded. In other words, calling a function does not affect the calling event's picking, even if *Copy picked* is enabled.

Asynchronous functions

A function can be set to *Asynchronous* (or *async* for short) in the [Add/Edit Function dialog](#). This allows it to be used with the *System Wait for previous actions to complete* action. This means if the function does any waiting itself, such as with an action like *Wait 3 seconds*, the caller can also wait for the function call to complete with *Wait for previous actions to complete*.

*Make sure the asynchronous function ends with a *Wait for previous actions to complete* action if it uses *async* actions. This ensures all *async* actions have been completed before the function itself is marked as having completed. Alternatively you may want to leave some *async* actions at the end that are not waited for, allowing the *async* function to finish without waiting for the last actions.*

Note this imposes a small performance overhead, so for best performance leave it disabled if you don't need it.

Function maps

Sometimes it's useful to be able to call a function depending on a string determined at runtime. The function maps feature allows for this. Try out the [Function Maps example](#) to see how it works.

Nesting and recursion

Like in programming languages, functions support calling functions from other functions, and functions calling themselves (recursion). Functions calling other functions or recursing create a new call stack entry with their own unique variables. In other words, like in programming languages, local variables and parameters are unique at each level of function call. This does not apply to static local variables or global variables.

Renaming the Functions object

When using functions, you'll notice actions and expressions refer to a built-in "Functions" object. This can be renamed in case you want to change it, but it does not appear in the Project Bar like other objects, so it has to be renamed a different way. There are two ways to rename it:

- 1 When adding an action, at the first step when you choose an object, right-click on *Functions* and click *Rename*.
- 2 Right-click on the name of the project in the Project Bar, and then choose Tools ► Rename Functions object. Note this will only appear if there are functions in your project.

JavaScript integration

Scripting

When using [scripts](#) in Construct, use `runtime.callFunction()` to call an event function from script.

External calls

In other cases, it is strongly recommended to use the [Addon SDK](#) to integrate JavaScript code with Construct. However it is possible to trigger a function from JavaScript using the following function:

```
c3_callFunction("name", ["param1", "param2"]);
```

Do not call this with Construct's scripting feature. It will not work correctly. In that context you must use `runtime.callFunction()` instead. This method only applies to external JavaScript.

The function with the given "name" is called by this method. Parameters are optional and can be omitted, but must be provided as an array in the second argument, and parameters may only be string, number or boolean values. The method also returns the return value set in Construct (if any), and also can only return a string or number.

If the project is running in a Web Worker with the Use worker setting, this method is still available on the DOM. However it instead returns a Promise resolving to the return value, and asynchronously calls the function by posting a message to the Web Worker. In an async function, `await c3_callFunction(...)` will always work.