1 the ampersand operator]] * [[#example_1:_jane_elizabeth_doe|2 example 1: jane elizabeth doe]] * [[#example_2:_jane_and_john_doe|3 example 2: jane and john doe]] * [[#example_3:_jane_and_john_go_to_the_market|4 example 3: jane and john go to the market]] ===== the ampersand operator ===== for first time users of visual3d's scripting language the use of the & operator may not be straightforward. the [[https://www.c-motion.com/v3dwiki/index.php?title=expressions|expressions]] page contains some links and examples of how to use this operator including: * [[https://www.c-motion.com/v3dwiki/index.php?title=expressions#using_pipeline_parameters_in_an_expression|using pipeline parameters in an expression]] * [[https://www.c-motion.com/v3dwiki/index.php?title=expressions#using_a_pipeline_parameter_as_part_of_a_signal_definition|using a pipeline parameter as part of a signal definition]] * [[https://www.c-motion.com/v3dwiki/index.php?title=expressions#specifying_string_data|specifying string data]] the ampersand (&) is used in pipeline commands to concatenate strings together, and is a separator for the parser to find pieces that need to be parsed separately. to get the most out of this tutorial, it is recommended that you run the commands in visual3d alongside reading the text descriptions, as the pipeline processing results dialog is a helpful tool for understanding and debugging in this interface. ===== example 1: jane elizabeth doe ===== let there be a visual3d workspace in which we have three parameters: first_name, middle_name and last_name. let us set each of these parameter values to be jane, elizabeth and doe, respectively: <code> select_active_file !/file_name=*.c3d ! /query= ! /subject_tags=no_subject ; set_pipeline_parameter /parameter_name=first_name /parameter_value=jane ; set_pipeline_parameter /parameter_name=middle_name /parameter_value=elizabeth ; set_pipeline_parameter /parameter_name=last_name /parameter_value=doe ; </code> say we want to create a new signal with the value jane elizabeth doe. we start by opening the pipeline command evaluate_expression: <code> evaluate_expression /expression=::first_name::middle_name::last_name /result_types=derived /result_folders=development /result_name=result ; </code> the output for running this command is **"error! parameter expression is not specified, and is a required parameter!".** if you run this, you will also notice that in the pipeline processing results dialog the expression field will be blank. when the expression field has been populated in the pipeline but appears blank in the results dialog, this typically indicates a syntax error. let's first clarify what the "::" does in this scripting language. using the parameter first_name, let us use evaluate_expression to see what first_name evaluates to: <code> evaluate_expression /expression=first_name /result_types=derived /result_folders=development /result_name=result ; </code> the expression evaluates to first_name, when what we wanted was jane. now if we add the "::" to our evaluate expression: <code> evaluate_expression /expression=::first_name /result_types=derived /result_folders=development /result_name=result ; </code> this evaluates to jane. this tells us that the "::" allows us to access the information stored under the variable first_name. recall earlier in this example, when the **"error! parameter expression is not specified, and is a required parameter!".** indicated a syntax error. we know that the "::" is correct because we want to access the data in the variables first_name, middle_name, and last_name. this is where the & operator comes in. the & operator acts as "glue" to stick multiple variables and strings together in the same expression. for example: <code> evaluate_expression /expression=::first_name&::middle_name&::last_name /result_types=derived /result_folders=development /result_name=result ; </code> gives the result: janeelizabethdoe. if you wanted to add spaces to this expression, you would use: <code> evaluate_expression /expression=&::first_name& &::middle_name& &::last_name /result_types=derived /result_folders=development /result_name=result ; </code> which now evaluates to jane elizabeth doe. ===== example 2: jane and john doe ===== say we want to compare two string variables a and b: <code> select_active_file /file_name=*.c3d ! /query= ! /subject_tags=no_subject ; set_pipeline_parameter /parameter_name=a /parameter_value="jane" ; set_pipeline_parameter /parameter_name=b /parameter_value="john" ; evaluate_expression

/expression=(::a=::b) /result_types=derived /result_folders=development /result_name=result ; </code> this expression successfully evaluates (meaning it gives no errors), but does not produce the correct result. as you can see in the image below, the expression explicitly evaluates (::a=::b) to a value of 0. it does not actually read in the variables that we want to compare. janeandjohn1.png let's add some "glue" and see what happens: <code> evaluate_expression /expression=(::a&=&::b) /result_types=derived /result_folders=development /result_name=result ; </code> this evaluates the expression jane = john as 1 or true...hmmm. practically speaking, we know this is not the case. this has occured because the set_pipeline_parameter function does not read the " " we assigned to these variables earlier. so to evaluate if these strings match, we have to add them back in: <code> evaluate_expression /expression=("&::a&"="&::b&") /result_types=derived /result_folders=development /result_name=result ; </code> this evaluates the statement jane = john to 0 or false. let's test to see if jane = jane is true: <code> evaluate_expression /expression=("&::a&"="&::a&") /result_types=derived /result_folders=development /result_name=result ; </code> and voila, there you have it. ===== example 3: jane and john go to the market ===== let’s say jane and john have gone to a fruit stand at a local farmer’s market. jane has purchased 33 apples and john has purchased 24 oranges. these values are stored in the derived signal type under the folder market as apples and oranges. let’s say you wanted to iterate through the market folder to find the total units of fruit that jane and john purchased. you could in this instance, simply call an evaluate expression for both and sum the totals. but if you have 5 types of fruit, or 20, it would be easier to calculate this parameter in a for loop. we start by setting all files to active and setting our total count to 0: <code> select_active_file /file_name=*.c3d ! /query= ! /subject_tags=no_subject ; </code> <code> evaluate_expression /expression=0 /result_types=derived /result_folder= market /result_name=total ; </code> because in this case the data we are dealing with is a variable within the workspace, not a pipeline parameter, we have to access it differently. to simply access the number of oranges we could use: <code> evaluate_expression /expression=derived::market::oranges /result_types=derived /result_folders=market /result_name=oranges ; </code> however, if we wanted to loop through these files, we again need to use the & operator. in the example below, the iteration parameter name is fruit and there are two iteration parameter items, apples and oranges. <code> for_each /iteration_parameter_name=fruit ! /iteration_parameter_count_name= /items=apples + oranges ; evaluate_expression /expression=derived::market::total + derived::market::fruit /result_types=derived /result_folders=market /result_name=total ; end_for_each /iteration_parameter_name=fruit ; </code> executing this example will cause the **“variable was not created because there were no results”** error. this is because of the way we accessed the fruit iteration parameter. because fruit is an iteration parameter, we need to access it as ::fruit to get the values of apples and oranges. however, because ::fruit will evaluate to apples or oranges, we need to “glue” and extra “::” before the iteration parameter: <code> for_each /iteration_parameter_name=fruit !/iteration_parameter_count_name= /items= apples + oranges ; evaluate_expression /expression=derived::market::total + derived::market&:&:&::fruit& /result_types=derived /result_folders=market /result_name=total ; end_for_each /iteration_parameter_name=fruit ; </code> this will correctly evaluate total to 57 fruits.