Conditional structures

Introduction

The conditional structures in Pine Script™ are if and switch. They can be used:

- For their side effects, i.e., when they don't return a value but do things, like reassign values to variables or call functions.
- To return a value or a tuple which can then be assigned to one (or more, in the case of tuples) variable.

Conditional structures, like the for and while structures, can be embedded; you can use an if or switch inside another structure.

Some Pine Script™ built-in functions cannot be called from within the local blocks of conditional structures. They are: alertcondition(), barcolor(), fill(), hline(), indicator(), library(), plot(), plotbar(), plotcandle(), plotchar(), plotshape(), strategy(). This does not entail their functionality cannot be controlled by conditions evaluated by your script — only that it cannot be done by including them in conditional structures. Note that while `input*.()` function calls are allowed in local blocks, their functionality is the same as if they were in the script's global scope.

The local blocks in conditional structures must be indented by four spaces or a tab.

`if` used for its side effects

An if structure used for its side effects has the following syntax:

```
if <expression>
    <local_block>
{else if <expression>
    <local_block>}
[else
    <local_block>]
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.
- <expression> must be of “bool” type or be auto-castable to that type, which is only possible for “int” or “float” values (see the Type system page).
- <local_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- There can be zero or more `else if` clauses.
- There can be zero or one `else` clause.

When the <expression> following the if evaluates to true, the first local block is executed, the if structure’s execution ends, and the value(s) evaluated at the end of the local block are returned.

When the <expression> following the if evaluates to false, the successive `else if` clauses are evaluated, if there are any. When the <expression> of one evaluates to true, its local block is executed, the if structure’s execution ends, and the value(s) evaluated at the end of the local block are returned.

When no <expression> has evaluated to true and an `else` clause exists, its local block is executed, the if structure’s execution ends, and the value(s) evaluated at the end of the local block are returned.

When no <expression> has evaluated to true and no `else` clause exists, na is returned.

Using if structures for their side effects can be useful to manage the order flow in strategies, for example. While the same functionality can often be achieved using the `when` parameter in `strategy.*()` calls, code using if structures is easier to read:

```
if (ta.crossover(source, lower))
    strategy.entry("BBandLE", strategy.long, stop=lower,
                   oca_name="BollingerBands",
                   oca_type=strategy.oca.cancel, comment="BBandLE")
else
    strategy.cancel(id="BBandLE")
```

Restricting the execution of your code to specific bars ican be done using if structures, as we do here to restrict updates to our label to the chart’s last bar:

```
//@version=5
indicator("", "", true)
var ourLabel = label.new(bar_index, na, na, color = color(na), textcolor = color.orange
if barstate.islast
    label.set_xy(ourLabel, bar_index + 2, hl2[1])
    label.set_text(ourLabel, str.tostring(bar_index + 1, "# bars in chart"))
```

Note that:

- We initialize the `ourLabel` variable on the script’s first bar only, as we use the `var` declaration mode. The value used to initialize the variable is provided by the label.new() function call, which returns a label ID pointing to the label it creates. We use that call to set the label’s properties because once set, they will persist until we change them.
- What happens next is that on each successive bar the Pine Script™ runtime will skip the initialization of

`ourLabel`, and the if structure's condition (barstate.islast) is evaluated. It returns `false` on all bars until the last one, so the script does nothing on most historical bars after bar zero.
- On the last bar, barstate.islast becomes true and the structure's local block executes, modifying on each chart update the properties of our label, which displays the number of bars in the dataset.
- We want to display the label's text without a background, so we make the label's background na in the label.new() function call, and we use `hl2[1]` for the label's *y* position because we don't want it to move all the time. By using the average of the **previous** bar's high and low values, the label doesn't move until the moment when the next realtime bar opens.
- We use `bar_index + 2` in our label.set_xy() call to offset the label to the right by two bars.

`if` used to return a value

An if structure used to return one or more values has the following syntax:

```
[<declaration_mode>] [<type>] <identifier> = if <expression>
    <local_block>
{else if <expression>
    <local_block>}
[else
    <local_block>]
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.
- <declaration_mode> is the variable's declaration mode
- <type> is optional, as in almost all Pine Script™ variable declarations (see types)
- <identifier> is the variable's name
- <expression> can be a literal, a variable, an expression or a function call.
- <local_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the <local_block>, or na if no local block is executed.

This is an example:

```
//@version=5
indicator("", "", true)
string barState = if barstate.islastconfirmedhistory
    "islastconfirmedhistory"
else if barstate.isnew
    "isnew"
else if barstate.isrealtime
    "isrealtime"
else
    "other"

f_print(_text) =>
    var table _t = table.new(position.middle_right, 1, 1)
    table.cell(_t, 0, 0, _text, bgcolor = color.yellow)
f_print(barState)
```

It is possible to omit the *else* block. In this case, if the `condition` is false, an *empty* value (`na`, `false`, or `""`) will be assigned to the `var_declarationX` variable.

This is an example showing how na is returned when no local block is executed. If `close > open` is `false` in here, na is returned:

```
x = if close > open
    close
```

`switch` structure

The switch structure exists in two forms. One switches on the different values of a key expression:

```
[[<declaration_mode>] [<type>] <identifier> = ]switch <expression>
    {<expression> => <local_block>}
    => <local_block>
```

The other form does not use an expression as a key; it switches on the evaluation of different expressions:

```
[[<declaration_mode>] [<type>] <identifier> = ]switch
    {<expression> => <local_block>}
    => <local_block>
```

where:

- Parts enclosed in square brackets (`[]`) can appear zero or one time, and those enclosed in curly braces (`{}`) can appear zero or more times.
- <declaration_mode> is the variable's declaration mode
- <type> is optional, as in almost all Pine Script™ variable declarations (see types)
- <identifier> is the variable's name
- <expression> can be a literal, a variable, an expression or a function call.
- <local_block> consists of zero or more statements followed by a return value, which can be a tuple of values. It must be indented by four spaces or a tab.
- The value assigned to the variable is the return value of the <local_block>, or na if no local block is executed.
- The `=> <local_block>` at the end allows you to specify a return value which acts as a default to be used when no other case in the structure is executed.

Only one local block of a switch structure is executed. It is thus a *structured switch* that doesn't *fall through* cases. Consequently, `break` statements are unnecessary.

Both forms are allowed as the value used to initialize a variable.

As with the if structure, if no local block is exectuted, na is returned.

`switch` with an expression

Let's look at an example of a switch using an expression:

```
//@version=5
indicator("Switch using an expression", "", true)

string maType = input.string("EMA", "MA type", options = ["EMA", "SMA", "RMA", "WMA"])
int maLength = input.int(10, "MA length", minval = 2)

float ma = switch maType
    "EMA" => ta.ema(close, maLength)
    "SMA" => ta.sma(close, maLength)
    "RMA" => ta.rma(close, maLength)
    "WMA" => ta.wma(close, maLength)
    =>
        runtime.error("No matching MA type found.")
        float(na)

plot(ma)
```

Note that:

- The expression we are switching on is the variable `maType`, which is of "input int" type (see here for an explanation of what the "input" form is). Since it cannot change during the execution of the script, this guarantees that whichever MA type the user selects will be executing on each bar, which is a requirement for functions like ta.ema() which require a "simple int" argument for their `length` parameter.
- If no matching value is found for `maType`, the switch executes the last local block introduced by `=>`, which acts as a catch-all. We generate a runtime error in that block. We also end it with `float(na)` so the local block returns a value whose type is compatible with that of the other local blocks in the structure, to avoid a compilation error.

`switch` without an expression

This is an example of a switch structure wich does not use an exppression:

```
//@version=5
strategy("Switch without an expression", "", true)

bool longCondition  = ta.crossover( ta.sma(close, 14), ta.sma(close, 28))
bool shortCondition = ta.crossunder(ta.sma(close, 14), ta.sma(close, 28))

switch
    longCondition  => strategy.entry("Long ID", strategy.long)
    shortCondition => strategy.entry("Short ID", strategy.short)
```

Note that:

- We are using the switch to select the appropriate strategy order to emit, depending on whether the `longCondition` or `shortCondition` "bool" variables are `true`.
- The building conditions of `longCondition` and `shortCondition` are exclusive. While they can both be `false` simultaneously, they cannot be `true` at the same time. The fact that only **one** local block of the switch structure is ever executed is thus not an issue for us.
- We evaluate the calls to ta.crossover() and ta.crossunder() **prior** to entry in the switch structure. Not doing so, as in the following example, would prevent the functions to be executed on each bar, which would result in a compiler warning and erratic behavior:

```
//@version=5
strategy("Switch without an expression", "", true)

switch
    // Compiler warning! Will not calculate correctly!
    ta.crossover( ta.sma(close, 14), ta.sma(close, 28)) => strategy.entry("Long ID'
    ta.crossunder(ta.sma(close, 14), ta.sma(close, 28)) => strategy.entry("Short ID
```

Matching local block type requirement

When multiple local blocks are used in structures, the type of the return value of all its local blocks must match. This applies only if the structure is used to assign a value to a variable in a declaration, because a variable can only have one type, and if the statement returns two incompatible types in its branches, the variable type cannot be properly determined. If the structure is not assigned anywhere, its branches can return different values.

This code compiles fine because close and open are both of the `float` type:

```
x = if close > open
    close
else
    open
```

This code does not compile because the first local block returns a `float` value, while the second one returns a string`, and the result of the `if` -statement is assigned to the `x` variable:

```
// Compilation error!
x = if close > open
    close
else
    "open"
```

Variable declarations

Loops