Control Flow - Part1: Conditional Statements

At a Glance

The If-Statement

Syntax:

```
if(condition)
{
    // things to do
}
```

Or:

```
if(condition)
{
    // things to do if condition is TRUE
}
else
{
    // things to do if condition is FALSE
}
```

Or:

```
if(condition1)
{
    // things to do if condition1 is TRUE
}
else if(condition2)
{
    // things to do if condition1 is FALSE, but condition2 is TRUE
}
else
{
    // things to do otherwise
}
string x;
```

Example:

```
string x;
// A simple conditional, with a single statement:
if(playerHasAllItems == true)
```

```
    GiveSanityBoost();

// A conditional with a code block:
if(playerHasAllItems)    // '== true' can be omitted, since
playerHasAllItems is a bool variable
{
    GiveSanityBoost();
    FadeOutAndTeleportPlayer();
}

// An if-else conditional:
if(itemsCollected < 5)
{
    GiveSanityBoostSmall();        // executed if itemsCollected < 5
}
else
{
    playerHasAllItems = true;      // executed otherwise
}

// Chained conditionals:
if(itemsCollected == )
{
    DisplayQuestMessage();        // executed only if itemsCollected is 0
}
else if(itemsCollected < 5)
{
    GiveSanityBoostSmall();       // executed if itemsCollected isn't 0, but
is less then 5
}
else
{
    playerHasAllItems = true;     // executed otherwise
}
```

The Switch-Statement

```
switch(itemsCollected)
{
case 1:
    GiveSanityBoostSmall();
    break;
case 2:
    GiveSanityBoostSmall();
    break;
case 3:
    GiveSanityBoost();
    break;
default:
    GiveSanityBoost();
    PlayVoiceFor(itemsCollected);
    break;
}
```

Discussion

The If-Statement

The if-statement, also known as the conditional statement, enables you to make a decision in your script, based on some condition. This is known as code branching; execution flow can follow one of these branches.
In it's simplest form, it has the following format:

```
// (this is the start of the if-statement)
if(condition)
{
    // things to do
}
// (the if-statement ends with '}')
```

Here, *condition* can be any expression that evaluates to a `bool` value - that is, any expression which can be either `true` or `false`. It can even be a function which returns a `bool`. After the if “header” line, there's a code block, which can be referred to as the body of the if-statement; this contains the code that is to be executed if the *condition* is satisfied (evaluates to `true`). If the condition is not satisfied, the whole body of the if-statement is simply skipped, and the execution continues immediately after it.

Example:

```
if(GetPlayerHealth() < 30.0f)
{
    StopMusic(6, );
    StopSound("amb_sound1", 6);
    StopSound("amb_sound2", 6);
}
```

Note: A code block (bounded by { and }) must be used if the body of the if-conditional contains more than one statement. Otherwise, only the first of those statements will be considered a part of the conditional (which then ends on the first `;`), while the others will execute no matter if the condition was true or not, since they will not be associated with it. For the same reason, if the body of the if-conditional consists of a single statement, you can omit the `{}`, but, for the sake of clarity, only indent that single line.

```
if(condition)
    singleStatement;    // executed only if condition == true
// (the if-statement ends with ';')
```

```
someOtherCode;      // executed regardles of the value of condition
// (NOTE: Make sure to decrease the indent, for the sake of clarity.)
```

If you prefer, you can choose to always use the code block notation ({}). This can help to avoid errors, and is a matter of preference.

Another form of the if condition allows you to specify two alternatives, one for each possible value of the *condition*:

```
if(condition)
{
    // executed only if condition == true
}
else
{
    // executed only if condition == false
}

// code continues here either way
```

You can also chain several if-statements: Another form of the if condition allows you to specify two alternatives, one for each possible value of the *condition*:

```
if(condition1)
{
    // executed only if condition1 == true
}
else if (condition2)
{
    // executed if condition1 == false, but condition2 == true
}
else    // <-- as usual, the else-branch is optional
{
    // executed if none of the above was true
}

// Basically, when chained, only one branch is executed:
// either the one that first matches the condition,
// or the final else-branch, if there was no match.
```

Note that to chain conditionals, you must use `else if`! If you don't the results will be different. The following pseudo-code looks similar, but it is *not the same*:

```
if(condition1)
{
    // executed only if condition1 == true
}

if (condition2)
```

```
{
    // executed if condition2 == true, REGARDLESS of condition1!
}
else
{
    // executed if condition2 == false
}

// In this example, there are TWO SEPARATE if-statements.
```

The Switch-Statement

The switch-statement is similar to a set of chained-if statements; it is used to pick one branch among several options, represented as integer values. Here's an example from the Amnesia game itself, from the script file for the first map; in this map, the Player awakens to the sounds of thunder and lighting flashes. The script emulates the visual and sound effects:

```
// Note: This was taken from the game itself; the code was reformatted to
match the style of this guide.

switch(GetLocalVarInt("ThunderStep"))
{
case 1:
    ThunderLights(2,0.05f);
    break;
case 2:
    ThunderLights(1,0.05f);
    break;
case 3:
    ThunderLights(3,0.05f);
    break;
case 4:
    ThunderLights(1,0.05f);
    break;
case 5:
    ThunderLights(2,0.05f);
    break;
case 6:
    ThunderLights(3,0.05f);
    break;
case 7:
    ThunderLights(1,0.3f);
    PlaySoundAtEntity("Thunder", "general_thunder.snt", "AreaThunder", 0.0f,
false);
    break;
}
```

In a more abstract representation, the format of the switch-statement is as fallows:

```
switch(inputInteger)
{
case CASE_CONSTANT1:
    // code to execute if inputInteger == CASE_CONSTANT1;
    break;
case CASE_CONSTANT2:
    // code to execute if inputInteger == CASE_CONSTANT2;
    break;
case CASE_CONSTANT3:
case CASE_CONSTANT4:
    // code to execute if inputInteger == CASE_CONSTANT3
    //              OR if inputInteger == CASE_CONSTANT4
    //               (this is why 'break;' is required)
    break;
default:
    // code to execute if nothing else matches (optional)
}
```

Each of the execution branches is bounded by `case CASE_CONSTANT#:` amd `break;`. If *inputInteger* equals some *CASE_CONSTANT#*, the code under that case is executed, until the first `break;` is reached. If you *forget* to place a `break;` where appropriate, you'll introduce suptle bugs to your script, so *be carefull*. For this reason, although it is possible to group several conditions together, as CASE_CONSTANT3 and CASE_CONSTANT4 were grouped in the pseudocode above, it is recommended that you avoid doing this. If two cases should execute the same code, extract that code into a function, and call that function for each case.

The final case doesn't require a `break;`, but it is a good practice to add one anyway, as you might forget to do it if you later on decide to change the code and add more cases below it.

Comparison Operators

- == → equals
- `!=` → not equals
- < → less then
- <= → less then or equal to
- > → larger then
- >= → larger then or equal to

Comparison operators are the usual mathematical operators you're familiar with. The result of their application is a `bool` (e.g. a < b is either `true` of `false`). This is why they can be used in if-statement and loop conditions. This also means that their result can be assigned to `bool` variables, like this:

```
bool allItemsFound = ( foundItems == totalItems );   // allItemsFound will
be either true or false
```

Important: Note that the comparison operator == (used to compare two values for equality) is *different* from the *assignment operator* = (which is used to assign values to variables). *Don't use the assignment operator* when checking for equality in your conditions - this is a common source of error. Always double check to make sure you typed in ==.

Using Logical Operators

The language also provides certain operators for use in `bool` expressions. These are NOT, AND, OR, and XOR operators; they enable you to form compound conditions, allowing you to express in code sentences like “If condition1 is true AND condition2 is true, then…”, or “If either condition1 OR condition2 is true, then…”, etc.

The box below lists the names of the operators, their keywords, an alternate symbolic C++-like notation, and their results for various value combinations of the operands.

```
// THE RESULTS OF THE BOOLEAN LOGICAL OPERATORS

// Notation:
//     true:  T
//     false: /

NOT (not, !):   // recommended to use '!' instead of 'not'
var     !var
--------------
 T        /
 /        T

AND (and, &&):
var1     var2      result
----------------------------
 T   and  T    ==    T
 T   and  /    ==    /
 /   and  T    ==    /
 /   and  /    ==    /

OR (or, ||):
var1    var2      result
----------------------------
 T   or  T    ==    T
 T   or  /    ==    T
 /   or  T    ==    T
 /   or  /    ==    /

XOR (xor, ^^):
var1     var2      result
----------------------------
 T   xor  T    ==    /
 T   xor  /    ==    T
 /   xor  T    ==    T
 /   xor  /    ==    /
```

The script language supports both keyword notation (not, and, or, xor), and symbolic notation (!, &&,

||, ^); keyword notation is more descriptive, and makes the condition read closer to an English language sentence, but the symbolic notation is more compact, and makes the variables stand out better, especially if there's no syntax highlighting support in your editor, or on a forum post. The choice is yours - pick one notation and stick to it; however, I recommend using the `!` instead of `not`, because it's short, and you usually write it together with the operand you're negating, making it clear to what part of the expression it applies. Also, it is consistent with the not equals `!=` comparison operator, and keeping your code consistent is a good thing.

As you can see, the NOT operator simply inverts the value of it's operand. Basically, you can use it to ask "If NOT something, then…". InPractice, it looks like this:

```
if (!enoughItems)   // read as "If NOT enoughItems"
{
    // omitted ...
}

// same as
if (not enoughItems)
{
    // omitted ...
}

// equivalent to:
if(enoughItems == false)
{
    // omitted ...
}
```

The AND operator is a binary operator (takes two operands), and it returns `true` *only* if both of the operands evaluate to `true`. This enables you to check for more than one condition simultaneously (and you can also chain several AND operators):

```
if (insideEventArea && enoughItems && !musicPlaying)  // read: If
insideEventArea AND enoughItems AND [NOT music playing])
{}

// same as:
if (insideEventArea and enoughItems and not musicPlaying)
{}
```

Since it's enough for one of the operands to be `false` in order for the AND operator to return `false`, if an operand which evaluates to `false` is found before the end of the compound expression, logical AND will just return `false`, and skip checking the remaining operands.

The OR operator is also a binary one, and it returns `true` if one or both of the operands are `true`. The only way for it to return `false` is if both operands are `false`. This is important to remember; the logical OR *does not* mean "either one OR the other (but not both)" - it's not exclusive (there's a different operator that is - the XOR operator). It means "any of the two (or both)". In fact, since all it cares about is for at least one of the operands to be `true`, if that operand happens to be the first one, the OR operator will return `true` immediatelly, without bothering to check the other one.

```
if (insideArea1 || insideArea2)     // read: If insideArea1 OR insideArea2
(or both - they could overlap)
{}

// same as:
if (insideArea1 or insideArea2)
{}
```

Finally, XOR is the *exclusive OR* operator. When one of the operands is `true`, and the other is not, the XOR operator returns `true`. Otherwise, when both operands are the same, it returns `false`. It's similar to OR, except that it “excludes” the case when both its operands are `true`.

```
if (insideArea1 ^^ insideArea2)     // read: If either insideArea1, OR
insideArea2 (but not both simultaneously)
{}

// same as:
if (insideArea1 xor insideArea2)
{}
```

When combining multiple expressions into one condition, use parentheses `()` to group things together and to explicitly define the order of the operations.

```
if ((foundItems == totalItems) && !(insideArea1 || insideArea2))   // read:
If [ found all items ] AND [ NOT [ insideArea1 OR insideArea2 ]]
{}
```