Constants and Enumerations

At a Glance

```
// Constants and enumerations allow you to define names which refer to values which
// should never change during the execution of the script. If you accidentally attempt
// to change them, you'll get a compiler error. They are also useful as a more meaningful
// and readable replacement for "magic numbers" - numerical values accepted as parameters
// by some functions, which have special meanings for those functions.

// Integer constants:
// The constants in this example define when should collision events take place;
// intended to be used with the AddEntityCollideCallback() engine function

const int COLLIDE_ON_ENTER = 1;
const int COLLIDE_ON_LEAVE = -1;
const int COLLIDE_ON_BOTH = ;

// Usage:
AddEntityCollideCallback("Player", "Area_Example", "ExampleCallback", false, COLLIDE_ON_BOTH);


// Some math constants:
const float PI = 3.1415926f;
const float E = 2.7182818f;



// Enumerated constants (enumerations) - often used as parameters to functions
enum Color     // Note: Enums are based on the int type.
{
    Red,        // has the default value of: 0
    Green,      // value: (previous + 1) = 1
    Blue        // value: (previous + 1) = 2, etc, if more added...
}


// Usage:
Color col = Color::Blue;  // emphasizes the containing enum, improves code clarity
```

```
// Or just:
Color col = Blue;          // same effect

// Assigning an integer value is not possible without an explicit
conversion:
Color col = 2;     // Causes compilation error!

// Converting from integers - should generally be avoided:
Color col = Color(2);   // Assigns Blue to col, since 2 corresponds to
Color::Blue

// However:
Color col = Color(-15);   // Has a value of -15, which is not tied to any
Color! Could be a problem!

// This is allowed:
int colValue = col;   // so, enums can be passed to functions expecting ints
--> see example below



// Enumerations - choosing your own values
enum CollisionState
{
    Leave = -1,       // = -1
    Both,             // =  0   (previous + 1)
    Enter             // =  1   (previous + 1)
}

// Usage:
AddEntityCollideCallback("Player", "Area_Example", "ExampleCallback", false,
CollisionState::Both);


// You can define all or some of the values; those left undefined will be
// assigned the value of previous_constant + 1
enum Ending
{
    Good = 1,           // =   1
    ReallyGood,         // =   2   (previous + 1)
    Neutral = -10,      // = -10
    ReallyBad = -2,     // =  -2
    Bad,                // =  -1   (previous + 1)
    Default = Neutral   // = -10    <-- You can use previously defined
constants
}

// Binary FLAGS:
enum QuestState
```

```
{
    FoundNote1   = 0x01,    // HEX for  1, which is in binary: 0000 0001
    FoundNote2   = 0x02,    // HEX for  2, which is in binary: 0000 0010
    FoundItem    = 0x04,    // HEX for  4, which is in binary: 0000 0100
    FoundPassage = 0x08     // HEX for  8, which is in binary: 0000 1000
    FoundAll     = 0x0F     // HEX for 15, which is in binary: 0000 1111
}
```

Discussion

Often, you'll need to define some values which are supposed to remain unchanged (constant) throughout the script, for the entire duration of its execution. You could use variables for this, being very careful not to change them, but even the best of scripters can make a mistake. The script language provides several constructs which enable you to define such constants, so that you can let the compiler worry about keeping them safe from you accidentally changing them. If you by any chance make a mistake and try to reassign a constant, the compiler will greet you with an error message - which is a good thing, because this protects your script from some potentially sneaky bugs.

Constants can be of any type, and are declared almost exactly the same way as variables; the only difference is that the declaration is preceded by the `const` keyword, and that the value has to be assigned immediately (because it cannot be changed later on).
const *typeName constantName* = *value*;

The names of the constants defined are often written using ALL_CAPS, and individual words are separated with an underscore _. This naming convention enables you to easily distinguish such constants from variables and other names, *but* it is a matter of preference, and is not required by the language.

It is common to define consts for mathematical and scientific constants, such are the numbers *pi*, *e*, or the gravitational acceleration *g*, etc. Other uses of constants include defining minimum or maximum values for something (max items, max health, etc...), or passing them as special values to functions (more on that later on).

```
const float PI = 3.1415926f;
const float E = 2.7182818f;

const int MIN_ITEMS_FOUND = 2;
const int MAX_ITEMS_FOUND = 5;

const float MAX_HEALTH = 100.0f;

const float MAX_WAIT_TIME = 10.0f;  // in seconds
```

You can use constants in various ways, for example, in conditions of conditional statements and loops, or in mathematical expressions, or you can assign them to variables (note: when assigning to a variable, the variable gets a copy of the value stored in the constant; the variable can then be manipulated without affecting the constant itself). For example:

```
float radius = 2.5f;
```

```
float circleArea = PI * radius * radius;

// OR
float health = MAX_HEALTH;

// later on...
health = 0.0f;
```

Constants As Function Parameters

A function can take a parameter to decide which way should it go about doing its job (see Functions - Part 1 to learn more about functions). If there are only two ways, the function might accept a `bool` parameter, and make a decision based on whether the value of that parameter is `true` or `false`. For example, the engine exposes this predefined function: `SetFogActive(bool abActive)`. It takes a single `bool` parameter, which it uses to decide if the fog should be activated or deactivated.

Often, though, there's more than two options, and in that case integers can be used to represent each of them. This approach is versatile in that it can be used to represent a varying number of options, however, the drawback is that the user of the function needs to remember the meanings of these numbers (sometimes referred to as “magic numbers”), and their meanings cannot be inferred just by reading the code.
Amnesia provides some functions which use special-meaning integer values as parameters. Such parameters often appear in callbacks as well. For example, this predefined function, which forces a lever to be stuck in a certain position, takes three parameters:

```
void SetLeverStuckState(string& leverName, int stuckState, bool useEffects);
```

The first parameter `leverName` tells it to which lever it should be applied to. The third parameter `useEffects` we can ignore in this discussion. The second parameter, `stuckState` is an `int` parameter, and it tells the function in which position the lever should be stuck. It can be either:

- -1 - stuck at “min” side
- 1 - stuck at “max” side
- 0 - not stuck, rests in the middle

This is how the function is used: if you wanted a lever which, say, controls the supply of electricity, called “Lever_Power”, to be stuck at the “max” position, which will mean that the power is on, you would write:

```
SetLeverStuckState("Lever_Power", 1, false);
```

As you can see, you can't tell, if you don't already know, what 1 means just by looking at that line of code - you have to consult the documentation. The number itself is essentially meaningless to a human reader. *Constants can help with this:*

```
// Globally declared constants (script-scope):
const int LEVER_MIN = -1;
const int LEVER_MAX = 1;
const int LEVER_FREE = 0;
```

```
//...

// Elsewhere in the program:
SetLeverStuckState("Lever_Power", LEVER_MAX, false);
```

Now it's clear that the the function makes the lever stuck in the “max” state. But, constants can do even better than that. It is still not obvious what this “max” state means in the context of the game (that is, your custom story or full conversion) itself. We assumed earlier that the lever is used to turn the power on or off. Thus, to convey this information, the code can be modified like this:

```
// Globally declared constants (script-scope):

// Leave these for any other, generic levers, which might
// possibly exist on the map
const int LEVER_MIN = -1;
const int LEVER_MAX = 1;
const int LEVER_FREE = ;


// Specific constants for the power-on/off lever:
const int LEVER_POWER_OFF = LEVER_MIN;
const int LEVER_POWER_ON = LEVER_MAX;


//...

// Elsewhere in the program:
SetLeverStuckState("Lever_Power", LEVER_POWER_ON, false);
```

Now it is fairly obvious what the call to SetLeverStuckState() function does.

Enumerations

Often you need to define a set of constants that are somehow related to each other, especially when they are intended to be passed as function parameters, like in the examples above. *Enumerations* provide a convenient language mechanism which enables you to group related constants together under one name. The syntax is as follows (in pseudocode):

```
enum EnumName
{
    EnumConstantNameA,
    EnumConstantNameB,
    EnumConstantNameC,
    EnumConstantNameD,

    // etc...

    // Note: it is not an error for the last
```

```
    // EnumConstant to be followed by a ',' symbol,
    // but it's generally not written
}
```

Use the `enum` keyword to declare an enumeration; then you give it a name, and simply list a bunch of constants between the `{` and `}`. As with variables, the names you chose for the enumeration and its constants should be meaningful to a human reader.
Enumerations (sometimes simply called enums) are based on the `int` type; that is, the values of the defined constants are nothing but integers under the hood. By default, the value of the first constant is , and all the other constants have the value of the previous constant incremented by one. So, using pseudocode from the code box above, the values are:

```
enum EnumName
{
    EnumConstantNameA,    //  = 0
    EnumConstantNameB,    //  = 1   (previous + 1)
    EnumConstantNameC,    //  = 2   (previous + 1)
    EnumConstantNameD     //  = 3   (previous + 1)
}
```

You are allowed to specify your own values for *some or all* of the enumerated constants. For those constants you didn't specify any value, the same (previous + 1) rule will be applied:

```
enum EnumName
{
    EnumConstantNameA = -2,    //  = -2
    EnumConstantNameB,         //  = -1   (previous + 1)
    EnumConstantNameC =  15,   //  = 15
    EnumConstantNameD          //  = 16   (previous + 1)
}
```

Enumerations, even though based on integers, should be viewed as code constructs which define a *new data type*, along with a set of values valid for the variables of that type. This is not entirely true, since variables of an enumerated type can be forced to store a value which is not defined in the corresponding list of enumerated constants, but let's ignore that for the moment. To declare such a variable, the same rules apply as for any other variable type:

EnumName variableName;

Just like with normal constants (those defined using the `const` keyword) declared at global (script-level) scope, enumerated constants can be accessed by their name. For example, to initialize a enum-type variable, you can write:

EnumName variableName = EnumConstantName;

However, for the reasons of code readability and clarity, when accessing enumerated constants it is often a good idea to include the name of the enumeration (*EnumName*) itself (especially when passing these values as parameters to functions), since *EnumName* often explains what is the common property used to group all the constants together in the first place. For this, the so-called scope resolution operator `::` is used:

EnumName variableName = *EnumName*::*EnumConstantName*;

For example:

```
enum Direction
{
    North,
    West,
    South,
    East
}

// Later on:
Direction sideOfTheWorld = Direction::North;
```

Another example:

```
// Let's assume there are only two types of items in your game

enum HealingPotion
{
    Small  = 10,
    Medium = 30,
    Large  = 60,
    Mega   = 100
}

enum BodyArmor
{
    Small  = 50,
    Large  = 100
}

// Later on:
HealingPotion vial = HealingPotion::Small;
HealingPotion leatherArmor = BodyArmor::Small;


// Or as a parameter to a hypothetical function
// void GivePlayerItems(HealingPotion potion, BodyArmor armor);

// Use:
GivePlayerItems(HealingPotion::Small, BodyArmor::Small);

// Rather than:
GivePlayerItems(Small, Small);   // which is which?

// Alternatively:
GivePlayerItems(vial, leatherArmor);
```

Relation to the Integer Type

As discussed, enums are based on the `int` type. This allows them to be implicitly converted to integers if required, so they can be used in place of them. This is especially useful with functions that use “magic numbers”. Implicit conversion *from* integers to enums, however, *is not allowed*.

```
enum PowerLeverState
{
    Undefined,      // = 0
    On,             // = 1
    Off             // = 2
}

// Later on:
// Accepts a value of type PowerLeverState, even though it expects an int
SetLeverStuckState("Lever_Power", PowerLeverState::On, false);

// Also:
int stateValue = PowerLeverState::Off;  // allowed; stateValue is now 2

// Wheres:
PowerLeverState state = 2;              // NOT allowed!
```

If you want to assign an `int` value to a enum-type variable, you have to explicitly convert it:

```
PowerLeverState state1 = PowerLeverState(2);    // now state1 =
PowerLeverState::Off

// Also:
int val = RandomInt(, 2);
PowerLeverState state2 = PowerLeverState(val);  // state2 takes a random
value
```

However, conversions like this should be done with care; although there are legitimate uses for them, conversions of this sort often indicate that your script can be better structured. When converting from integers, you run a risk of making your enum-type variables contain values other than those that have been defined as acceptable for their type:

```
PowerLeverState state1 = PowerLeverState(100);  // state1 = 100. Now what?


// Also:
int val = -1;

// later on
PowerLeverState state2 = PowerLeverState(val);  // state2 = -1. Could this
cause problems?
```

Although this compiles, and no error is displayed, depending on how your script is written, this may or

may not cause problems. As conversions of this kind might happen accidentally, if your script relies on enums always having valid values, then there's a great chance that it will fail if a problematic enum-typed value comes along. So, always assume that such a value could be passed to your functions, and decide what to do if it happens. Often enough, you can simply ignore that case, and make your function do nothing and just silently end, but, if the underlying integer value is somehow used (for example, to cycle through all the possible values in a given enumeration), you need to check if the value provided is in the valid range first, and replace it with some acceptable value if it's not.

Using Enums to Define Binary Flags

Coming Soon... See “At a Glance”