Variables

At a Glance

```
int x;
int a, b;
int c = 2, d;
float pi = 3.14f;
bool isAmnesiaTheBest = true;
string subtitle = "The Dark Descent";
```

Discussion

The concept of variables is pretty much the same as used in math – a variable is something that can take on a number of different values. To be more precise, variables represent a scripter-friendly way to access locations in computer's memory – because we scripters are humans, it is easier for us to declare a variable with a meaningful, human-readable name, than to poke around the internal workings of the system. You can use variables to store various types of data.

In order for it to be used in your code, a variable must first be *declared*.
A *variable declaration* is where you introduce the variable for the first time, thus making it "known" to the script engine.

A declaration has the following format:
typeName variableName;
or
typeName variableName = initialValue;

For example:
`int unlitCandlesLeft = 4;`

Here, the *type* of the variable is integer, which is denoted by the `int` keyword. The name of the variable is `unlitCandlesLeft`, and it's set to the initial value of 4. The scripter chooses the variable name; it should reflect the purpose of the variable. Using such descriptive names makes the code more readable. Here, for example, `unlitCandlesLeft` could be used to count how many candlesticks in a level the Player should light up before some gameplay event happens. Variable names *cannot start with a number*.

The script language supports different *variable types*, but for the purposes of Amnesia map scripting, it is enough to get familiar with just four basic data types:

- `int` – represents integer numbers (... -4, -3, -2, -1, 0, 1, 2, 3, 4...)
- `float` – represents floating-point (real) numbers (like 3.14, 0.001, -12.0, 1.41)
- `string` – represents text; string literals are always placed in quotes ("Amnesia", "HPL2 Engine")
- `bool` – represents Boolean values (can be either `true` or `false`)

You choose the type for your variable depending on what kind of data you want to store. You wouldn't

choose `int` if you want to store text. The feature of the language that enforces type constraints on variables is called *type safety*. This is important for several reasons. First, it can help debug your code if problems arise. Next, implicit conversions are generally not allowed among all types. Also, operations on variables of different types yield different results. For example, if you add two `int` numbers, you get the mathematically correct result:
2 + 3 = 5

If you add two `string`-s, each of which is a textual representation of a number, the operation results in a new string which is a combination of the original two (string *concatenation*):
“2” + “3” = “23”
“Amne” + “sia” = “Amnesia”

Time to learn how to use variables in code. Building on the Quick Start example, let's begin with the following code for our script:

```
void OnEnter()
{
    FadeOut(0.0f);  // Turns the screen black instantly

    // Now that the screen is black, do a 5 second fade in
    // for dramatic effect.
    FadeIn(5.0f);
}
```

Let us now add two global variables, and use them to represent the durations for the screen fade effects. We will pass these variables to the fading functions in place of the numerical constants:

```
float fadeOutTime = 0.0f;   // in seconds
float fadeInTime;           // (uninitialized)

void OnEnter()
{
    FadeOut(fadeOutTime);  // Turns the screen black

    // Now that the screen is black, do a 5 second fade in
    // for dramatic effect.

    // Assign a value to the fadeInTime variable
    fadeInTime = 5.0f;  // in seconds
    FadeIn(fadeInTime);
}
```

First, two variables are declared. The variable `fadeOutTime` is also initialized to the value of `0.0f` (the 'f' suffix tells the compiler that the literal 0.0 should be treated as a `float`; without the suffix, such numerical values are considered a different type, `double`, which is similar to `float`, but it can be used to represent a wider range of values. If the 'f' suffix is omitted, an implicit conversion from `double` to `float` is made, if possible. It's a good practice to use the 'f' suffix, although it's often not required).
The `fadeOutTime` variable is then passed in as a parameter to the FadeOut() function, in place of the hard-coded number from the original code. This is possible since variables contain values, assigned to

them by the scripter.

The second variable, named `fadeInTime`, is left uninitialized for demonstration purposes. The aim of the code is to show you how you can also (re)assign values to variables in other places in the script. The line
`fadeInTime = 5.0f;`
assigns the value of `5.0f` to the variable. After that, the variable is used as a parameter to the FadeIn() function.

What is the advantage of this?
Variables can be declared and initialized in one place, and then be reused in several other places in the code. Changing the value of the variable will affect all the code that uses it. In programming, it is a good practice to avoid redundant repetitions; when the same thing is repeated many times in many places, it is easy to make an error, but it is hard to fix it. Variables offer one way of organizing your code to avoid such repetition.

Global vs Local Scope

Another point of interest is *where* the variables are declared. In the previous example, they were declared *globally*, outside of any function. *Global variables* are accessible from every part of your script file (from every function). If a variable is declared *inside the body* of a function, it is called a *local* variable. *Local variables* are available for use only from within that same function (and only in lines that come after the one where the variable is declared). Trying to use such a variable from a different function would result in an error.

```
void OnEnter()
{
    float fadeOutTime = 0.0f;   // in seconds
    float fadeInTime = 5.0f;    // in seconds

    FadeOut(fadeOutTime);  // Turns the screen black

    FadeIn(fadeInTime);  // Fade in for dramatic effect.
}
```

In the code above, the two variables from previous example are now made local to the OnEnter() function. They can be used in pretty much the same way as before, but *only from within* that function. If you attempt to use them from a different function, you'll get an error message – to test this, add the OnLeave() function listed below, and then quick-reload the map.

```
void OnEnter()
{
    float fadeOutTime = 0.0f;   // in seconds
    float fadeInTime = 5.0f;    // in seconds

    FadeOut(fadeOutTime);  // Turns the screen black

    FadeIn(fadeInTime);  // Fade in for dramatic effect.
}
```

```
void OnLeave()
{
    // The next line should result in a compilation ERROR!
    fadeInTime = 1.0f;
}
```

The error says:

```
INFO: Compiling void OnLeave()
ERR : 'fadeIn Time' is not declared
```

This error is in the context of the OnLeave() function; the script compiler reached that function, and then the line where `fadeInTime` was used (where the attempt was made to assign the value of `1.0f` to it). However, since no global variable with that name exist in this script, and no local declaration was found either, the compiler issues an error. So, the meaning of the error message is: although `fadeInTime` was declared somewhere in the file, the variable is local to a different function – it was not declared *neither globally, nor inside OnLeave()*, where it was used.

To fix the error, delete (or comment out) the problematic line of code. The script below compiles and works.

```
void OnEnter()
{
    float fadeOutTime = 0.0f;   // in seconds
    float fadeInTime = 5.0f;    // in seconds

    FadeOut(fadeOutTime);  // Turns the screen black

    FadeIn(fadeInTime);  // Fade in for dramatic effect.
}

void OnLeave()
{
}
```

Since the engine exposes the SetGlobalVar*Typename*() & SetLocalVar*Typename*() functions (and their Add- and Get- counterparts) for seting *game variables*, where the terms global and local have a slightly different meaning than what was presented here for *script variables*, here's a table that explains how these predefined functions and script-declared variables compare:

// Global game variables ➢ SetGlobalVarInt() ➢ SetGlobalVarFloat() ➢ SetGlobalVarString()	**Global-scope (Game-scope)**	Visible across multiple maps (that is, multiple map scripts).

// Global script variables
 	
➢ int maxTries = 3;
 	
// Local game variables
 	
➢ SetLocalVarInt()
➢ SetLocalVarFloat()
➢ SetLocalVarString()
	Script-scope (Map-scope)	Visible in one map script, from all functions.

// Local script variables
// (local-scope declarations
// and function parameters)
 	
➢ void OnTimerElapsed(string timerID)
➢ {
➢ int index = 0;
➢ // code using index omitted...
➢ }
	Local-scope (code block scope, function scope)	Visible only within a code block which contains their declarations (code block is bounded by { and }, e.g., in functions.)

The functions which manipulate global game variables, SetGlobalVar*Typename*(), AddGlobalVar*Typename*() and GetGlobalVar*Typename*() [where *Typename* can be either Int, Float, or String], provide a way for you to store and use a value in a storage location which is maintained for the duration of your game (or custom story). It can be accessed from more than one map script, and any changes made in one map will be visible from the others. They are global *relative to the maps*.

Functions which manipulate local game variables, SetLocalVar*Typename*(), AddLocalVar*Typename*() and GetLocalVar*Typename*(), allow you to define and use storage locations for the current map script. They are visible from every part of the map script, once set for the first time. So they are *local to that particular map script* (can't use them from a different map), but are *global relative to functions* in that script. In this respect, they are similar to the script variables declared globally (outside any function, or code block).

Simultaneous Declarations

It is possible to declare more than one variable in a single line, by separating variable names with commas. For example, the code
`float x, y;`
declares two variables of the same type, float, named x and y.

You can also initialize some or all of them in such a declaration:

```
int a = 1, b = 2;
int x = 5, y;
```

Using Variables to Help Others (And Yourself!)

Those of you who write code snippets, or even longer scripts, for your fellow forum members can use variables to make your code easier to use and understand. Often, the same name or value appears in several places. Also, the script usually contains a lot of placeholder values, and when the person you are helping takes over the code, a lot of these placeholder values need to be replaced by the ones which are adequate in the context of their custom story or full conversion. The replacement process can be tedious and error prone, but variables come to the rescue (although search & replace tools can mitigate the problem).

Instead of hard coding the names and values, you can assign them to variables at the beginning of your script, and then use variables instead. This way, if any changes need to be made, it only needs to happen in *one place* - at the start, where the variable declarations are. This eliminates the need to dig through the code, trying to find every single appearance of some value.

Also, remember to give your variables descriptive names, so that others (and you) can quickly figure out what are they used for! You can indicate where the placeholder values should be replaced with a comment or two.

```
// Example: Fictional Help-Script
// ---------------------------------

// Replace these assignments to match the names in your map
string scriptArea_jumpScare1 = "Area_JumpScare1";
string scriptArea_jumpScare2 = "Area_JumpScare2";
string scriptArea_startMusic = "Area_StartMusic";
string scriptArea_lookAtTarget = "Area_LookAt";
string particleSys_Orbs = "PS_Orbs";

// You can adjust these values as well, if you want
float screenFadeInTime = 6.0f;
float musicFadeInTime = 3.5f;

// ...

// Here goes the code that uses the variables above (omitted).
// Note: the script would use a variable wherever the corresponding value
would otherwise appear. E.g.:
// FadeIn(screenFadeInTime);         // ---> instead of the hard-coded
FadeIn(6.0f);

// ...
```

Naming Convention Tips

Variable names should be meaningful – one should be able to infer from their name what are they used for. These are some examples of good variable names:

```
int numTinderboxes;
int jumpScaresLeft;
float fadeOutDuration;
bool isDoorLocked = true;
```

In contrast, bad names for the same variables would be:

```
int tndrs;
int jpsCnt;
float fod;
bool lckd;
```

Most people start variable names with a lowercase letter. If a variable name is composed of more than one word, you can use the so called camelCasing-based notation to separate individual components of the name. This is what you'll encounter throughout this guide. Another option is to use an underscore, like this:

```
int num_tinderboxes;
int jump_scares_left;
```

You can also combine the two:

```
string area_jumpScare1;
string area_jumpScare2;
string area_playerStart;
```

Which approach to use is entirely your choice, but once you decide upon one, it is a good idea to be consistent and stick to it. This will make it easier for you and other people to read and understand your code.

See Also: Types.