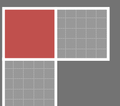
N001

Nuklear LORD Rewrite

Software Architecture – NukleairScript Syntax

Syntax guide for the Nuklear LORD Game Creation System, including all information needed to create a game world or a modification to a game world.
Version 0.3

Matt Buchwald
Nuklear LORD Development Team
2/26/2009



Revision History

Version	Date	Description
0.3	2/26/2009	Initial Draft Review

OVERVIEW 6

INITIAL DESIGN 6

VARIABLES 7

VARIABLE TYPES 7
DECLARING VARIABLES 7
ASSIGNING VALUES TO VARIABLES 7
VARIABLE SCOPE 8

OPERATIONS 9

OPERATOR ORDER OF EVALUATION 9
ARITHMETIC OPERATIONS 10
COMPARISON OPERATORS 10
BITWISE OPERATIONS 11
BITSHIFT OPERATORS 11
BITWISE AND, BITWISE OR, BITWISE XOR, AND BITWISE NOT 12
LOGICAL AND, LOGICAL OR, AND LOGICAL NEGATION 12
EXPRESSIONS 13

PLAYER STATS, MONSTER STATS, AND CONFIGURATION 14

PLAYER STATS 14
DECLARING PLAYER STATS 14
ACCESSING PLAYER STATS 14
CREATING A NEW PLAYER 15
LOADING PLAYER STATS 15
SAVING PLAYER STATS 16
ENEMY STATS 16
MONSTER STATS 16
DECLARING MONSTER STATS 16
ACCESSING MONSTER STATS 17
DEFAULT MONSTER STATS 17
LOADING MONSTER STATS 18
CONFIGURATION 18

DECLARING CONFIGURATION PARAMETERS........19
ACCESSING CONFIGURATION PARAMETERS........19

CONTROL STRUCTURES........21

GROUPS OF STATEMENTS: THE BLOCK........21
THE CONDITIONAL STRUCTURE: THE IF STRUCTURE........21
THE IF STATEMENT........21
THE ELSE STATEMENT........22
MULTIPLE CONDITIONS WITH IFS........23
THE ITERATIVE STRUCTURE: THE WHILE LOOP........23
THE ITERATIVE STRUCTURE: THE FOR LOOP........24

FUNCTIONS........26

DECLARING FUNCTIONS........26
EXAMPLES OF FUNCTIONS........26
CALLING FUNCTIONS........28

LOCATIONS........29

DECLARING LOCATIONS........29
LOCATION PROPERTIES........29
MENU ITEMS........30
ADDING MENU ITEMS........32
LOCATION TESTS AND ACTIONS........33
VALIDITY TESTS........34
ENTRY ACTIONS........35
EXIT ACTIONS........36
LOCATION ACTIONS........36
ENTERING A LOCATION........36
LEAVING A LOCATION........36
LOCATION EXECUTION ORDER........37

GAME WORLD STRUCTURE........38

A SIMPLE SAMPLE GAME........38
GAME FLOW ORDER OF EXECUTION........42
MAINTENANCE ACTIONS........43

GATEWAYS AND RANDOM SELECTIONS 44

GATEWAYS 44
RANDOM SELECTION STRUCTURES 46

Overview

This specification will be used during the development of Nuklear LORD Software to track and document the major architecture decisions of the *NuklearScript* Syntax. Software development is a fluid process and it is not practical to pre-engineer the system down to every component prior to writing any code. It is more useful to design guidelines and an overall philosophy to the code structure to give the development and final code consistency and clarity.

As the software is developed, this document will grow and capture the design decisions made during the various phases of the project. Overviews of these phases are listed next.

Initial Design

In the initial design, we present the initial syntax plan for the *NuklearScript* scripting language.

Variables

Several different variable types are available in NuklearScript.

Variable Types

Variable	Meaning	Range
short	16-bit integer	-32768 to 32767
int	32-bit integer	-2147483648 to 2147483647
long	64-bit integer	-9223372036854775808 to 9223372036854775807
double	Double Precision Floating Point	Any real number. No limit to range, but accuracy decreases as numbers get very large or very small.
bool	Boolean	false or true
string	String Variable	A string of ASCII characters.
char	Character Variable	A single ASCII character.

Declaring Variables

Variables make up the data building blocks of the any language. In *NuklearScript*, all variables must be declared before they can be used. A variable declaration consists of the variable type, followed by the desired variable name and then a semicolon. Variable names must begin with a letter and may contain letters, numbers, and underscores. Examples:

```
int damage;

bool lightShield;
```

This would declare one 32-bit integer variable named damage and one Boolean variable named lightShield.

Assigning Values to Variables

Once declared, variables may be assigned a value using the assignment operator (**=**). The default value for all numeric variables is zero. Boolean variables default to false while string variables default to the empty string (""). Examples:

```
damage = 500;

lightShield = true;
```

The above example would set **`damage`** equal to 500 and **`lightShield`** equal to true. The value to the right of the assignment operator may be a value, an expression, or a function that returns the correct type of value.

You may also combine declaration and assignment to initialize a variable to a given value. Examples:

`int damage = 500;`

`bool lightShield = true;`

Variable Scope

All variables exist only within the function in which they are declared. If you wish a variable to be used in another function, you must pass it as a parameter to that function.

For Example:

```
function void Scope1()
{
	int num1;
	int num2;
	string string1;
	string string2;

	num1 = Random(1,10);
	num2 = Random(1,10);

	string1 = SomeOtherFunction();

	while(num1 > num2)
	{
		num2 = num2 + 1;
		string2 = AnotherFunction(num1, num2);
	}
}
```

The function **`SomeOtherFunction`** cannot access variables **`num1`**, **`num2`**, **`string1`**, or **`string2`** because they are defined outside its scope, even though **`SomeOtherFunction`** is called from within **`Scope1`**. **`AnotherFunction`** can access the value of **`num1`**and **`num2`**because they have been passed to this function as parameters. The **`while`** loop can access **`num1`**, **`num2`**, **`string1`**, or **`string2`** because it is a control structure (not a function) that is contained within **`Scope1`**, where those variables are declared. Functions will be discussed in depth in a later section.

Operations

NuklearScript shall support a wide range of arithmetic, logical, and bitwise operations, following a strict order of evaluation, as described below.

Operator Order of Evaluation

Operator(s)	Operation(s)	Order of Evaluation
()	Grouping	Evaluated first. If the parentheses are nested, the innermost pair is evaluated first. If there are multiple pairs at equal nesting levels, the pairs are evaluated left to right.
!, ~, -	Logical Inversion, Bitwise Inversion, Negation	Evaluated second.
*, /, %	Multiplication, Division, Modulus	Evaluated third. If there are several, they are evaluated left to right.
+, -	Addition, Subtraction	Evaluated fourth. If there are several, they are evaluated left to right.
<<, >>	Bitshift Left, Bitshift Right	Evaluated fifth. If there are several, they are evaluated left to right.
&	Bitwise AND	Evaluated sixth. If there are several, they are evaluated left to right.
^	Bitwise Exclusive OR	Evaluated seventh. If there are several, they are evaluated left to right.
|	Bitwise Inclusive OR	Evaluated eight. If there are several, they are evaluated left to right.
&&	Logical AND	Evaluated ninth. If there are several, they are evaluated left to right.
||	Logical OR	Evaluated tenth. If there are several, they are evaluated left to right.
<, <=, =>, >	Inequality Comparisons	Evaluated eleventh. If there are several, they are evaluated left to right.
==, !=	Equality Comparisons	Evaluated twelfth. If there are several, they are evaluated left to

		right.
=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Assignment	Evaluated last. If there are several, a compiler error occurs.

Arithmetic Operations

Use arithmetic operations to add, subtract, divide, and multiple values in *NuklearScript* code. *NuklearScript* also provides the modulus operator (`%`), which yields the remainder of an integer division.

Each arithmetic operator is a binary operator, which means they take two operands. For example, the expression **`integer1 + integer2`** takes two operands, **`integer1`** and **`integer2`**, and one binary operator **+** and returns their sum. This operation can be part of a larger expression or used alone in an assignment operation.

For example, to multiply **`a`** and **`b`** and assign the resulting product to **`c`**, we write:

```
c = a * b;
```

NuklearScript applies all operators in a strict order determined by the *rules of operator precedence* described in the table at the beginning of this section. For example, multiplication, division, and the modulus operator are all applied before the addition and subtraction operators. If you wish to apply operators in a different order, parentheses may be used to group operations much like they are used in algebraic expressions. For example, the result of **`2 * (1 + 3)`** is **8**. Since **`1 + 3`** is contained within parenthesis, it is applied before the multiplication operator. If the parentheses were removed, the result would instead be **5**.

Comparison Operators

Use comparison operators to compare two values. Comparison operators are comprised of the equality operators **==** and **!=** and the relational operators **> >= < <=**.

The following function demonstrates the functionality of the comparison operators.

```
function void ComparisonExample()
{
	int num1;
	int num2;

	num1 = Random(1,10);
	num2 = Random(1,10);

	if (num1 = num2)
	{
		DisplayText num1 + " is equal to " + num2;
```

```
        }
        if (num1 != num2)
        {
                DisplayText num1 + " is not equal to " + num2;
        }
        if (num1 < num2)
        {
                DisplayText num1 + " is less than " + num2;
        }
        if (num1 > num2)
        {
                DisplayText num1 + " is greater than " + num2;
        }
        if (num1 <= num2)
        {
                DisplayText num1 + " is less than or equal to " + num2;
        }
        if (num1 >= num2)
        {
                DisplayText num1 + " is greater than or equal to " + num2;
        }
}
```

This function will generate two random numbers from 1 to 10, and then apply the vaious comparison operators. Some example results:

```
4 is not equal to 9
4 is less than 9
4 is less than or equal to 9
```

```
6 is not equal to 2
6 is greater than 2
6 is greater than or equal to 2
```

```
9 is equal to 9
9 is less than or equal to 9
9 is greater than or equal to 9
```

Bitwise Operations

Use bitwise operations to modify an integer value at the bit level.

Bitshift Operators

The Bitshift Left and Bitshift Right operators (**<<** and **>>**) shift each bit in an integer to the left or right a number of places equal to the second operand. For example, **`integer1 >> 1`** will shift each bit of **`integer1`** to the right by one place. This is equivalent to dividing **`integer1`** by 2. Shifting to the left by 1 is equivalent to multiplying by 2. To see this, let's say that **`integer1`** is equal to 12.

In binary, **`12 = 0000 1100`** and if we shift every bit to the right by one, we get **`0000 0110`**, which equals 6.

Bitwise AND, Bitwise OR, Bitwise XOR, and Bitwise NOT

The Bitwise AND operator performers a logical AND on each bit of an integer with the corresponding bit of another integer. This can be used to pull out specific bits of an integer to see if they are set or cleared. For example, the following code tests to see if bit 3 of **`integer1`** is set. Recall that 8 is **`0000 1000`** in binary, and the rightmost bit is bit 0.

```
if (integer1 & 8 > 0) // Bit 3 of integer1 is set
{
     //Things to do when bit 3 is set.
}
```

As seen in the table of operator order of evaluation, the relational operators have lower precedence than the bitwise operators, and will always be performed afterwards.

You can test multiple bits at the same time, however doing so in the example code above will only guarantee that at least one of the masked bits is set. For example, a bitwise and of 12 will test for bits 3 and 2 and will return true if either is set.

The bitwise OR operator performs a logical OR on each bit of an integer with the corresponding bit of another integer. This can be used to set a specific bit of an integer. For example, the following code sets bit 5 of integer1. Recall that 32 is **`0010 0000`** in binary and the rightmost bit is bit 0.

```
integer1 = integer1 | 32;
```

You can set multiple bits at once with this command. A bitwise OR of 40 will set bits 5 and 3.

The Bitwise XOR operator performs a logical XOR on each bit of an integer with the corresponding bit of another integer. The resulting bit is set if and only if both bits corresponding bits are at a different logical state. For example, **`8 ^ 12 = 4`**, since both numbers have bit 3 set, but only one has bit 2 set.

The Bitwise NOT operator performs a logical inversion on each bit of an integer. For example, ~254 will return the value 1 since **`254 = 1111 1110`** in binary. Inverting all the bits yields **`0000 0001`**. This can be combined with a Bitwise AND to clear a specific bit. The following code clears bit 3 of **integer1**.

```
integer1 & ~8 > 0); // Clear bit 3
```

Logical AND, Logical OR, and Logical Negation

Use these logical comparisons to compare two Boolean expressions. Logical AND will return true if and only if both expressions are true, while logical OR will return true if either is true. Logical negation will invert the truth value of a Boolean expression.

Examples:

`(3 > 4) && (4 > 5) = false AND true = false`

```
(6 < 10) && (7 != 8 ) = true AND true = true

(3 > 4) || (4 > 5) = false OR true = true

(6> 10) || (7 == 8) = false OR false = false

!(3 > 4) && (4 > 5) = !false AND true = true AND true = true
```

Logical AND and Logical OR have higher precedence than the relational and equality comparisons, so be sure to use parentheses to surround relational and equality comparisons when combining them with logical comparisons.

Expressions

Expressions are a group of operators, and may include function calls. Expressions may be a simple expression, which is comprised of one operator and its operands , or more complicated expressions with multiple combinations of simple expressions. As an example:

```
a = (b + 1) / (c * tan(x, y));
```

This shows a complex expression with multiple groupings and a function call. The result of the expression is assigned to the variable **a**. Note that **a** itself and the assignment operator that follows are not part of the expression.

Player Stats, Monster Stats, and Configuration

Stats make up the bulk of character and creature interaction with the game world. There are three types of stats: Player Stats, of which there are two objects, Monster stats, and Configuration.

Player Stats

Player stats are the statistics that describe a player character.

Declaring Player Stats

Player stats make up the core of the player character's definition. As such, the game world creator will need to be able to declare stats for the player. Player stats are declared at the top level of code, meaning they are declared outside of all functions. While they may be declared anywhere in a file outside of functions, it is best to do so at the beginning of a file. If your module uses a stat that is defined in another module (or the main game world) that will be loaded, you do not need to declare it. However if you attempt to use a stat that has not been declared, the configuration application will throw an error. If you attempt to declare a player stat with the same name, but a different type than an already declared stat, the configuration application will also throw an error. So long as a stat has the same variable type, it can be declared multiple times without error.

```
playerstat type stat_name;
```

To declare a player stat, type the keyword **`playerstat`**, followed by the stat's variable type and the stat's name. The stat's name is case-sensitive.

Accessing Player Stats

In order to access a player stat, use the **`Player`** object. The **`Player`** object is a special *NuklearScript* keyword that represents that player's stats.

```
Player.stat_name
```

The keyword **`Player`**, followed by the dot operator (which has the highest precedence), followed by the stat name allows the programmer to set or retrieve the stat. Let's see an example:

```
playerstat bool Alive;
playerstat int MaxHp
playerstat int Hp;
playerstat bool Male;

function void DivineIntervention() {
      Player.Alive = true;
      Player.Hp = Player.MaxHp;
      if (!Player.Male){
            DisplayText "Brave lady, be revived!";
      }
      else {
```

```
            DisplayText "Brave sir, be revived!";
      }
}
```

Here, you see a set of four player stats declared outside of any function. In the **DivineIntervention** function, we set two player stats, **Alive** and **Hp**. **Alive** is set equal to **true**, a value. In the next line, you can see that player stats are use just like variables, and can be t he identifier that receives a new value, or used in an expression to act as a value. Lastly, you can see the fourth stat used in a Boolean expression.

Creating a New Player

All **Player** objects have, by default, the following stats:

Stat	Type	Meaning
ID	int	Player ID number. Unique for each player. If a player is new, this is set to zero.
LoginName	string	Player's login name. Unique for each player.
Online	bool	Online status of this player. Set **true** once the **enter** function is called (more on that later.)
New	bool	New status of this player. This is **true** whenever the player's login name is not present in the game database.

When a player logs into the game, his **Player** object is automatically populated with default values for all other variables declared by the **playerstat** command. As such, no direct action is required to create a new player object.

If a player is new after entering the game using the **EnterGame** function, the game will call the **NewPlayer** function, which will generate the player's **ID**. You can use the **NewPlayer** function to populate the default starting stats for a player. This will be discussed in depth later on.

Loading Player Stats

All non-default stats (meaning everything except **ID**, **LoginName**, **Online**, and **New**) can be loaded from the database using the **Player** object's **Load** function. This will cause all non-default stats to immediately be replaced by their values that are stored in the database, so make sure any stats you wish to remain unchanged are saved before calling this function. It has a relatively simple syntax:

```
Player.Load;
```

This command should always be called shortly after entering the game inside the **EnterGame** function.

Sometimes, however, you only need to load one or a few stats. You can instead save an individual stat using the **Player** object's **LoadStat** function.

```
Player.LoadStat.stat_name;
```

This statement will replace the specified stat with the value stored in the database.

Saving Player Stats

You can save all non-default stats by using the **Player** object's **Save** command.

```
Player.Save;
```

All stats will immediately be saved to the database when this command is invoked. This should always be called at the end of the **NewPlayer** function to ensure the initial stats are saved.

If you wish to only save a single stat, use the **Player** object's **SaveStat** command.

```
Player.SaveStat.stat_name;
```

This will save the specified stat to the database immediately.

Enemy Stats

The **Enemy** object is a special kind of player object that cane be loaded with any player character's stats (including the current player's, if that's useful for some reason). All **Player** object functionality is the same in the **Enemy** object, with the following (slight) differences:

The **LoadStat**, **Save**, and **SaveStat** functions all operator based on the **ID** currently stored in the **Enemy** object. The **Load** function takes a slightly different syntax:

```
Player.Load id_num;
```

The **Load** function requires the id number of the player whose stats you wish to load into the **Enemy** object. This will load all non-default stats, as well as the **ID** and **Online** stats.

Monster Stats

Player stats are the statistics that describe a monster.

Declaring Monster Stats

Monster stats make up the details of creatures your players' characters face. Monster stats are declared at the top level of code, meaning they are declared outside of all functions. While they may be declared anywhere in a file outside of functions, it is best to do so at the beginning of a file. If your module uses a stat that is defined in another module (or the main game world) that will be loaded, you do not need to declare it. However if you attempt to use a stat that has not been declared, the configuration application will throw an error. If you attempt to declare a monster stat with the same name, but a different type than an already declared stat, the configuration application will also throw an error. So long as a stat has the same variable type, it can be declared multiple times without error.

```
monsterstat type stat_name;
```

To declare a player stat, type the keyword **monsterstat**, followed by the stat's variable type and the stat's name. The stat's name is case-sensitive.

Accessing Monster Stats

In order to access a monster stat, use the **Monster** object. The **Monster** object is a special *NuklearScript* keyword that represents the currently loaded monster's stats.

```
Monster.stat_name
```

The keyword **Monster**, followed by the dot operator (which has the highest precedence), followed by the stat name allows the programmer to set or retrieve the stat. Let's see an example:

```
monsterstat int Hp;
monsterstat int Strength;
monsterstat string Name;

function void MonsterBattle() {
      int damage;
      if (Random(10)==7) {
            DisplayText Monster.Name + "delivers a powerful strike!";
            damage = PowerMove(Monster.Strength);
      }
      ..
      //More code here that eventually lets player deal damage…
      ..
      DisplayText "You deal " + damage + " damage to " + Monster.Name;
      Monster.Hp -= damage;
}
```

Here, you see a set of three player stats declared outside of any function. In the **MonsterBattle** function, we use some Monster stats as values, and set another one. Monster stats are used just like variables.

Default Monster Stats

All monsters have the following stats:

Stat	Type	Meaning
ID	int	Monster ID number. Unique for each monster in a specific group.
Level	int	Level of the monster.
Group	string	Group the monster belongs to.
Frequency	Int	Rarity of the monster. Higher numbers are less rare.

Each monster in a given Group has a unique ID number. Monsters are arranged by Group, Level, and ID. These stats may not be modified in-game.

Loading Monster Stats

Loading a monster's stats can be done one of two ways: Randomly, or specifically. To load a specific monster, we use the following:

```
Monster.Load group_name, id_num;
```

This will load the monster from the group *group_name* that has the ID number *id_num* into the **Monster** object. An example:

```
function void LoadMaster() {
      Monster.Load Masters Player.Level;
}
```

This will load a monster from the **Masters** group that has an ID number equal to the player's level.

Sometimes we just want to load a range of monsters. In this case we use the following:

```
Monster.Load group_name, low_level, high_level;
```

This will load a random monster from the group *group_name* that has a level from *low_level* to *high_level*. The frequency stat of the monster is used to weight which monsters are more likely to appear. An example:

```
function void LoadMonster() {
      if (Random(1,10)==7) {
            Monster.Load Forest 1 Player.Level;
      }
      else {
            Monster.Load Forest Player.Level Player.Level;
      }
}
```

This will have a 1 in 10 chance to load a random monster from the **Forest** group anywhere from monster level 1 to a monster level equal to the player's level. Otherwise it will load a random monster from the **Forest** group of an equal level with the player.

In all load cases, if no valid monster exists for the given load parameters, then all monster stats will be initialized to the default values for their variable types.

Configuration

Configuration parameters are statistics that are global to the game. These may be **readonly**, meaning they may only be modified only by the configuration application, or they may be **normal**, meaning they may be modified in-game.

There are no default configuration parameters.

Declaring Configuration Parameters

Configuration parameters make up the details of your game world. Configuration parameters are declared at the top level of code, meaning they are declared outside of all functions. While they may be declared anywhere in a file outside of functions, it is best to do so at the beginning of a file. If your module uses a parameter that is defined in another module (or the main game world) that will be loaded, you do not need to declare it. However if you attempt to use a parameter that has not been declared, the configuration application will throw an error when loading the game world modules. If you attempt to declare a configuration parameter with the same name, but a different type than an already declared parameter, the configuration application will also throw an error. Likewise if the read-only/normal status of the parameter differs, the configuration application will throw an error. So long as a parameter has the same variable type and read-only/normal status, it can be declared multiple times without error.

Configuration parameters may be declared in one of two ways:

```
configuration readonly type parameter_name "DisplayName"[ = value];
```

```
configuration normal type parameter_name "DisplayName"[ = value];
```

Configuration parameters marked by the **readonly** keyword may be modified by the configuration application, but not within the game. If the parameter is marked by the **normal** keyword, then the initial value may be edited by the configuration application, while the current value is modified from within the game. The initial value is set when the *Reset Game World* option is chosen within the configuration application. Both forms may optionally set the default value by putting an equal sign followed by the default value. The Display Name is the name of the parameter as seen in the configuration application.

Accessing Configuration parameters

In order to access a configuration parameter, use the **Config** object. The **Config** object is a special *NuklearScript* keyword that represents the currently loaded monster's stats.

```
Config.stat_name
```

The keyword **Config**, followed by the dot operator (which has the highest precedence), followed by the stat name allows the programmer to set or retrieve the stat. configuration parameters that are **readonly** may be used as values, but may not be assigned to. Let's see an example:

```
configuration readonly int ForestFights "Forest Fights Per Day" = 15;
configuration normal int CurrentDay "Current Game Day" = 0;
function void NewDay() {
	Player.ForestFights = Config.ForestFights;
	Player.LastDayPlayed = Config.CurrentDay;
	..
	Config.ForestFights = 25; //This would trigger an error!!
}
```

In this example, two configuration parameters are created. The first is a read-only parameter called `ForestFights`. It has a default value of 15. This default value is used the first time the parameter is created in the application. From then on, the parameter can be edited to any value from within the configuration application, and will never reset to 15 unless the module that created this parameter is removed, and the *Clean Configuration* option is chosen in the configuration application.

The second parameter is a normal parameter called `CurrentDay`. This has a default value of 0. When the parameter is first created in the application, this parameter will be set to have a current value of 0. This default value can then be edited in the configuration application so that next time the game world is reset, it will use the configured value instead of this default value.

Control Structures

Games would be very boring if they had a straightforward, predictable sequence that never altered, so that's why *NuklearScript* offers several control structures in order to modifying and direct program flow.

Groups of Statements: The Block

A statement is a single line of a command that ends with a semicolon. Sometimes we would line to group these commands together; to do that we use a block. A block begins with an opening brace and ends with a closing brace. A block appears as though it were a single statement, even though it may contain multiple statements.

```
//Everything between { and } is in the block.
{
	int temp;
	temp = Random(1,10);
	if (temp >7) DisplayText "You win!";

}
```

Variables declared within a block have a scope within that block. Once the block has completed execution, any variables declared within the block are removed.

The Conditional Structure: The If Structure

The **`if`** statement is your basic conditional statement. In essence, this statement says, "If some Boolean expression is true, then do these actions."

The If Statement

```
if (condition) statement
```

The basic **`if`** structure has the form shown above. The *`condition`* in parentheses must be a Boolean expression. A Boolean expression is any expression that evaluates to a Boolean value: **`true`** or **`false`**. The expression may be a single Boolean variable, or a more complex expression involving Logical ANDs and relational operators. When the *`condition`* evaluates to **`true`**, the *`statement`* is executed. If the expression evaluates to **`false`**, the next statement after the **`if`** structure is executed instead.

```
bool success;
//Some other logic would go here that would set bool to true or false.
if (success) Player.Gold += 500;
DisplayText "Thank you for playing!";
```

In the above example, if the variable **`success`** is **`true`**, then the player will gain 500 gold, and then the program will move on and display "Thank you for playing!" However, if the variable **`success`** is **`false`**, then the program will move past the statement belonging to the **`if`** structure and simply display "Thank you for playing!"

In most cases, you will want to do more than one thing if a condition is true. In that case, a block may be used to form a multi-step statement. Remember that blocks appear as though they are single statements. With this in mind, we might instead do something like this:

```
bool success;
//Some other logic would go here that would set bool to true or false.
if (success)
{
      Player.Gold += 500;
      DisplayText "Arrrgh, you got lucky, kid!";
}
DisplayText "Thank you for playing!";
```

In the above example, the program will also display "Arrrgh, you got lucky, kid!" when the **success** variable is **true**.

It is a good programming practice to always use a block with an **if** structure, even if you only need to execute one line of code. This will make it clear that you intend to only execute that line of code and will make debugging simpler.

The Else Statement

Sometimes you want to do something else when the condition is **false**, and only when the condition is false. The else statement says, "If the condition wasn't true, do this instead."

if (***condition***) *statement*

else *statement*

Whenever the *condition* is **true**, it will execute the *statement* belonging to the **if**, but if the *condition* is **false**, it will execute the statement belonging to the **else** instead. If the condition is true, then the **else's** *statement* will never execute. With this in mind, we might further modify the example to do something like this:

```
bool success;
//Some other logic would go here that would set bool to true or false.
if (success) {
      Player.Gold += 500;
      DisplayText "Arrrgh, you got lucky, kid!";
}
else {
      Player.Gold -= 500;
      DisplayText "Haha! Better luck next time, kid.";
}
DisplayText "Thank you for playing!";
```

Now, when the player is unsuccessful, he will lose gold and be taunted. How cruel!

Multiple Conditions with Ifs

Sometimes multiple conditions exist at the same time, each requiring a separate action. You can chain if statements onto else statement to process multiple conditions at once.

```
int result;
//Some other logic would go here that would set the result.
if (result > 95) {
      Player.Gold += 1000;
      DisplayText "Wow! I've never seen anyone so skilled!";
}
else if (result > 50) {
      Player.Gold += 500;
      DisplayText "Arrrgh, you got lucky, kid!";
}
else if  (result > 10) {
      Player.Gold -= 500;
      DisplayText "Haha! Better luck next time, kid.";
}
else {
      Player.Gold -= Player.Gold;
      DisplayText "Ha! You really suck, kid. So I'm taking it all.";
}
DisplayText "Thank you for playing!";
```

As you can see, we now have multiple conditions with multiple results. Notice that we've been putting the opening brace for the block on the same line as the **`if`** statements. Since a statement begins with a valid dentifier (keyword, variable name, etc) or an opening brace and ends with a semicolon or a closing brace, the *NuklearScript* system uses those to determine where the statement begins, so we are free to use as much (or as little!) whitespace as we desire. Whitespace includes extra spaces and any carriage returns and linefeeds such that you get when you hit the enter key. It is recommended that you start a block on a new line, but for space considerations we will often not do this for our examples.

The Iterative Structure: The While Loop

Sometimes you will want to repeat a set of statements multiple times. To do this, we use an iterative structure such as the while loop.

```
while (condition) statement
```

The **`while`** loop will execute the *`statement`* while the *`condition`* is true. The *`condition`* must be a Boolean expression. If the *`condition`* is **`false`** upon initially reaching the **`while`** loop, then its *`statement`* will never be executed. Like the **`if`** structure, it is generally good programming practice to always make the statement a block.

We might use the while loop to do something like this:

```
int answer = Random(1,20);
int guess = 0;
```

```
short counter = 0;

DisplayText "I'm thinking of a number from 1 to 20!";
DisplayText "Try to guess it and I'll let you know how long it took!";
while (guess != answer) {
      counter += 1;
      guess = GetTextInput(2);
}
DisplayText "It took you " + counter + " tries to guess my number!";
```

This code will loop over and over while the player enters up to two characters in a line of text, until the player's guess matches the right answer. The condition here won't ever be **true** initially because the **guess** is initialized to 0 and the **answer** will never be zero.

The Iterative Structure: The For Loop

The for loop is similar to the while loop in that it will repeat a statement multiple times, but differs in that it contains an initialization statement and a modification statement. It is thus ideally suited to perform a repetitive action with a counter that is initialized and increased on each action.

```
for (initialization; condition; modification) statement
```

The *initialization* is a statement used to initialize a counter variable associated with the **for** loop. The *condition* is the Boolean expression that must be **true** in order for the next pass of the loop to execute. The *modification* is a statement used to modify the counter variable.

Before the **for** loop is executed, the system executes the *initialization* statement. At the beginning of each loop, the *condition* is checked. If **true**, the *statement* is executed. If the condition is not **true**, the *statement* is not executed and the program continues with the next statement after the for loop. At the end of a loop, the *modification* statement is executed.

```
damage = 0;
DisplayText "Enemy attacks with a Quad-Strike Blow!";
for (int i = 0; i < 4;i += 1) {
      damage += Random(0, Enemy.Str -1 ) + Enemy.Str - 1;
}
DisplayText "Enemy hits you for " + damage + " damage!!";
Player.Hp -= damage;
```

Here we have a monster attack that hits four times for the monster's regular damage. Since we know we want to repeat the damage calculation four times, we use a **for** loop. First, the *NuklearScript* system creates and initializes the count variable **i** to 0. Then it checks if **i** is less than 4. If it is, it performs a pass through the loop. During the loop, the **damage** variable is increased by some damage calculation. Lastly the counter variable **i** is increased by one. After the fourth pass through the loop, **i** will be equal to 4, so the condition will no longer be true. Thus, this loop will only execute four times.

It should be noted that the initialization and modification can be any kind of statement, including blocks, and the condition may be any Boolean expression. With this in mind, you can create some very creative

constructs should the need arise, such as a battle that lasts only four rounds, or until one opponent suffers fatal damage. Perhaps there is also a poisonous gas that automatically lowers the player's strength every round the battle goes on?

Functions

Functions are the backbone of the *NuklearScript* system. Almost all functionality is performed by executing functions. Functions are code units that may take zero or more parameters and return one or zero values.

Declaring functions

Functions are declared using the function keyword. Functions must be declared at the top level of code, meaning that you cannot declare a function within another function.

```
function type FunctionName(type paramName1, ...) statement
```

A function declaration begins with the keyword **`function`** followed by the return value type of the function. If a function does not have a return type, the **`void`** type is used. Next, the name of the function follows. The function name must be unique and it is case sensitive, meaning that correct upper and lowercase characters must be used every time the function is called. For example, **`Function1`** is a different function than **function1**. After the function name, a list of parameters is enclosed within parentheses. Each parameter declaration is composed of a variable type followed by the parameter name. Each parameter may be used like a variable within the function, however keep in mind that any modification done to them will not be done to the original variable passed into the function. Parameter are said to be passed *by value*. If a function has no parameters, an empty set of parentheses is used. After the parameter list, the statement that the function executes is found. This is nearly always a block.

Some example functions declarations, with their statements left blank for the time being:

```
function void DisplayCharmString(short charm) { ... }

function int GetBaseDamage(short strength) { ... }

function void NewDay() { ... }

function long Power(int base, int exponent) { ... }
```

Examples of Functions

As you can see, a wide variety of functions is possible. Let's look at a more in-depth function declaration.

```
function int GetBaseDamage(short strength) {
     return Random(0, Enemy.Str -1 ) + Enemy.Str - 1;
}
```

Here we have a function that takes one short integer as a parameter and returns an integer. This function takes as a parameter a strength value, and returns the base damage of a regular attack. Notice that the function itself calls another function called **`Random`**, which is a built-in function of the *NuklearScript* system. The **`return`** keyword is used to designate the value that will be returned by the

function, and ends the function execution immediately. Now let's look at a different function declaration.

```
function void NewDay() {
	Player.Alive = true;
	Player.Hp = Player.MaxHp;
	Player.ForestFights = Config.ForestFightsPerDay;

	//Some other stuff goes here

	if (!Player.Male){
		if (Player.Lays > 0 && Player.Lays > 5 * Player.Kids)  {
			if (Random(1, 18) == 18)         {
				string gender;
				if (Random(2)==0) gender = "boy";
				else gender = "girl";
				DisplayText "You give birth to a baby " + gender;
			}
		}
	}

	//More code would go here

	if (Player.Inn) {
		DisplayText "You awake, well rested from your stay.";
	}
	else {
		DisplayText "You wake up in the field, well rested.";
	}

	Player.LastDayPlayed = Config.CurrentDay;
}
```

Here we have a function intended to handle the start of a new day in our game world. The function in question does not return a value and takes no parameters. Instead, it merely modifies the player's data. You can also see a variety of nested conditional statements, which determine the actions that will be performed.

This function has no return statement. If a function has no value to return, a return statement is unneeded. However, if you wish to stop a function's execution early, you may use a return statement to do so, as in this example:

```
function void DragonSlain() {
	if (!Player.Alive) { //Player is dead, so don't do anything.
		return;
	}
	//Other code would follow, but is omitted for the example.
}
```

As you can see, this function would normally perform more actions, but in the event that it is run when the player is dead, function immediately ceases instead.

Calling Functions

Now that we know how to create functions, we can use them in other pieces of code, including other functions. A function must always be called with a parameter list, even if it has none. Functions with no parameters are called with an empty set of parentheses.

Functions with no return type may be called as a single statement, as in this example:

```
function void WriteLogHeader() {
      int rand = Random(3);
      if (rand == 0) {
            WriteLog("The village is quite. No children are playing.");
      }
      else if (rand == 1) {
            WriteLog("You can feel the terror the beast inspires.");
      }
      else if (rand ==2) {
            WriteLog("The tears of the widows fills the streets.");
      }
      DrawFancyLine();
}
```

Inside this function, we have a function call to DrawFancyLine. Since it is a void function, it is executed as a single statement. DrawFancyLine takes no parameters, so we call it with an empty set of perentheses. We also have several calls to WriteLog, which is also a void function. As it takes a string as a parameter, we pass in a sting.

Functions that return a value may be called as part of an expression, as an expression themselves, or on a single line. Functions that are parts of an expression act as the value type they return. If a function returns a bool, then it may be used in any location a bool value is expected, whereas if it return an integer, it may be used in any location an integer value is expected.

Examining the previous example, we see a function call to **Random**, a built-in function of the *NuklearScript* system. Since the **Random** function returns an integer, it is used to initialize the value of the variable **rand**. Here, it is used to determine a random value from 0 to 2 (a range of 3 values). In the first function example in the *Functions* Section, you can see **Random** being used in a more complex expression.

As we mentioned, you can also call a function that has a return type as its own statement. In such a case, the return value is discarded. You might do something like this in a function that takes two parameters, displays some information or performs a calculation based on them, and then returns status code that may be positive or that may indicate an error. In some cases, you might want to check that error. In other cases you may simply want to perform the function’s actions and do not care about the result.

Locations

Locations serve as places, and are generally used to make menus in a game that represent a location, such as a town square.

Declaring Locations

Declaring a **location** is done similarly to that of declaring a variable. One key difference is that locations must be defined at the top level of code, outside of all functions.

```
location location_name;
```

The above code will add the **location** *location_name* to the available locations. A **location** must be declared before it can be used. A **location** may be declared multiple times, though subsequent declarations are ignored.

Location Properties

Now having a location is useful, but obviously a simple declaration is not sufficient. Obviously locations need to know what keys will trigger various menu items, how to display its menu, and other things. Each location has a set of properties that can be defined.

Property	Type	Meaning
Name	string	Name of the **location**.
Prompt	string	Prompt generated for this **location**.
Keys	string (read-only)	Generates a comma-seperated list of all valid, non-hidden key commands for this **location**.
SimpleMenu	Universal string Function	String containing the menu for this **location**. Displayed only if **MenuFile** is an empty string.
MenuFile	string	File containing the menu for this **location**
MenuSection	string	Subsection of the **MenuFile** that contains the menu for this **location**.
Tests	Universal Test Function	Contains a list Universal Testing Functions that determine if the **location** should be entered.
Entry	Universal Function	Contains a list of Universal Functions that are executed upon entry into this **location**.
Exit	Universal	Contains a list of Universal Functions that are executed upon

	Function	leaving this **location**.
Actions	Universal Function	Contains a list of Universal Functions that are executed continuously while waiting for a key press in this **location**.
Menu	menuitem	Contains a list of **menuitems** that represent all possible selections from this **location**.

Each of these properties is set at the top level of code (usually right after the declaration). It is easiest to show some of them by example:

```
location useMage;
useMage.Name = "** MAGICAL SKILLS **";
useMage.SimpleMenu = useMage_menu;
useMage.Prompt = useMage_prompt;

function string useMage_menu()
{
     return "`%" + useMage.Name + "\n\n`0" + Enemy.Name +
            "`2 looks confused as you sheath your " +
            Player.WeaponName + ".";
}

function string useMage_prompt()
{
     return "`5You have `%" + Player.MageUses +
             "15Use Points. Choose. [`#Nothing`5] : ";
}
```

At this point, the menu does no actions, and has no way to know what to do when you hit a key. However, we have set a few properties. We set the **Name** property, which is meant to be used as a display name for this location. In the very next line, we set the **SimpleMenu** property, which is used to create the menu display for the location. This takes a Universal String Function as its value, which is a function that returns a string and takes no prarameters. We also set the **Prompt** property to set the prompt that will be displayed after the menu and menu items are displayed.

Menu Items

Speaking of menu items; we don't have any yet. Now is a good item to introduce the **menuitem**, which has the following properties:

Property	Type	Meaning
Key	char	Key pressed to use this option.
Text	string	Text used to display this menu option. (Only if the menu uses a **SimpleMenu**).

DisplayTest	UniversalTest	A universal test function to determine if the menu item should be displayed (**SimpleMenu** only);
PromptTest	UniversalTest	A universal test function to determine if the menu item's key should appear in the list of keys. (in a location's **Keys** property)
AllowTest	UniversalTest	A universal test to determine if the menu item will be accepted.
Refresh	bool	True if a location should re-enter the location upon returning from this command, thus executing all entry actions. Defaults to **true**.
Actions	Universal Function	The functions that this menu item will execute.

We create a **menuitem** using the following declaration:

```
menuitem gateway_name;
```

Like **locations**, this is declared at the top level of code and may be declared multiple times. Only the first declaration is used. Let's create a one in the following example:

```
menuitem useMage_V;
useMage_V.Key = 'V';
useMage_V.Text = "`5(`#V`5)anish                           (`%4`5)";
useMage_V.Refresh = false;
useMage_V.DisplayTest = useMage_V_Display;
useMage_V.PromptTest = false;
useMage_V.AllowTest = useMage_V_Allow;
useMage_V.Actions += DoSkillVanish;
useMage_V.Actions += LeaveLocation;

function bool useMage_V_Display()
{
      return Player.MageSkill >= 4 && Player.MageUses >= 4;
}
function bool useMage_V_Allow()
{
      return Player.MageUses >= 4;
}
function void DoSkillVanish(){
      Player.Vanish = true;
}
```

Here we create a **menuitem** called **useMage_D**. It is good form to label the menu items by the name of the menu they will belong two, followed by an underscore and then the key this item represents. Next we assign the key that this menu represents, using the **Key** property of the **menuitem**. We then set the prompt text we will use when displaying this item (only if the location's **MenuFile** property is

not set) using the **Text** property. Finally, we set the **Refresh** property to false, so that upon return from this command, no entry actions will be done (which is trivial, since there are none).

Next, we set the **DisplayTest** property to equal the **useMage_D_Display** function. Now, whenever this menu item is displayed by the location, it will only appear in the display if **useMage_D_Display** returns a true value. In this case, when the player's skill points and mage uses are both greater than or equal to 4.

We set the **PromptTest** property equal to **false**. This means that the **location**'s **Keys** property will never list the key. We could also set it to **true** to make it always show the location. In effect, this means that these tests can accept a universal testing function or a **bool** value. All three kinds of tests behave this way, and default to **true**;

Lastly, we add the **DoSkillVanish** function to the actions, as well as the **LeaveLocation** function. The **Actions** property is a list of Universal Functions (void functions with no parameters) that are execture when this **menuitem**'s **Key** is pressed in the **location**. We add functions to the **Actions** property by using the **+=** assignment operator:

menuitem_name.**Actions** += *action_name;*

The above statement will add the function *action_name* to the actions of *menuitem_name*. You can also remove an action using the **-=** assignment operator instead of the **+=** assignment operator.

The **LeaveLocation** function is a built-in function of *NuklearScript* that informs the game that we will leave most recent location once we return to it from the menu item's execution.

Adding Menu Items

Now that we can create menu items, we can add them to the **location**'s menu. Before we do that, however, it should be noted that the **Menu** property of a **location** has a few properties of its own.

Property	Type	Meaning
Columns	int	Number of columns to use in a simple menu. Defaults to 1.
Padding	int	Width of a column. If positive, will add extra spaces to the right. If negative, will add extra spaces to the left. Defaults to 20.
Alignment	int	If zero, will center menu items on the screen. If negative, align the menu items to the right. If positive, will align menu items to the left. Defaults to 1.

These properties define how the menu will display its items when the location is using the **SimpleMenu** and not the **MenuFile**.

menuitems can be added to a menu by using the **+=** assignment operator. Simply tack them on to the **Menu** in the order you wish them to be listed. The system will display them in the assigned order in both the simple menu display and in the **location**'s **Keys** property.

location_name.**Menu** += *menuitem_name;*

The above statement will add an item to the menu. You can also remove a menu item (a module may want to do this for some reason) by using the above statement with the **−=** assignment operator instead of the **+=** assignment operator. Let's revisit the useMage location.

```
useMage.Menu.Columns = 1;
useMage.Menu.Padding = 30;
useMage.Menu.Alignment = 1;

useMage.Menu += useMage_F;
useMage.Menu += useMage_V;
useMage.Menu += useMage_I;
useMage.Menu += useMage_L;
useMage.Menu += useMage_S;
useMage.Menu += useMage_M;
```

Here we've added six menu items to our **Menu**, representing the various skills. Depending on skill amounts, we then might see the following when our player enters the **location**:

```
** MAGICAL SKILLS **

Massive Giant looks confused as you sheathe your Death Sword.

(F)lick Real Hard                (1)
(V)anish                         (4)
(I)nferno Strike                 (8)

You have 10 Use Points. Choose. [Nothing] :
```

Location Tests and Actions

You may have noticed that our example does not use the **Entry**, **Exit**, or **Actions** properties. A simple location, such as the one we created for casting spells, probably won't need those. However, more traditional location uses, such as taverns or caverns will probably want to check things such as a player being dead, or not having some requirement to be there. Let's create a tavern.

```
location bcTavern;
bcTavern.Name = "The Bloodcrow Tavern";
bcTavern.Prompt = bcTavern_prompt;
bcTavern.MenuFile = "dwModuleMenus.txt";
bcTavern.MenuSection = "BCTAV";

function string bcTavern_prompt() {
     return "`5" + bcTavern.Name + "`7   (? for menu)\n" +
```

```
                "`7(" + bcTavern.Keys + ")\n\n" +
                "`2Your desire, `0" + Player.Name +
                "`2? [`%" + Config.TimeLeft + "`2] : ";
}
```

Here we've created a little tavern. We use the section BCTAV from the dwModuleMenus.txt file as our menu, so no **SimpleMenu** property needs to be set. This will display everything after @#BCTAV and everything before the next @# combination in that file and use it as the menu's display. The prompt will be shown as normal. Let's say that we add some menu items later on, since they aren't important for this discussion. Let's also say it's a pretty rough place with mean patrons that requires you to have a special pass to enter. How might we do this?

Validity Tests

Let's start with the validity test. Validity tests are used to test whether a player can enter or remain at a location. Each validity test is a **bool** function with no parameters. If any validity test returns **false**, the player will not enter (or remain in) the **location**. All validity tests will run, regardless of whether or not a previous test failed or passed.

location_name.**Tests** += *test_name;*

The above statement will add the validity test *test_name* to the tests of *location_name*. You can also remove a validity test using the **-=** assignment operator instead of the **+=** assignment operator.

```
bcTavern.Tests += bcTavern_CheckPass;
function bool bcTavern_CheckPass() {
      if (Player.HasBctPass) {
                  DisplayText "Looks like you're legit, kid.";
            }
      }
      else {
            DisplayText "Scram kid. You're not wanted here.";
            return false;
      }
}
```

Now, whenever we enter the tavern, it'll check for the player's pass. If he has it, they'll let him know he's legit, and the function will return true, allowing this test to pass. If not, they'll let him know he's not wanted and return false, resulting in the player being kicked out of the location.

We have one problem, however: Since validity tests are run constantly while waiting for a key press, the player will be spammed with hundred of "Looks like you're legit, kid" messages. We obviously don't want this behavior and luckily there is a way around it.

The **Player** object has a special property called **CurrentLocation** that returns the name of the current **location**. When first trying to enter the location, we are not yet in the **location**. Therefore the first time we run the validity tests, we aren't yet in the bcTavern **location**. We can therefore use this code to only display text on the initial validity test:

```
bcTavern.Tests += bcTavern_CheckPass;
function bool bcTavern_CheckPass() {
      if (Player.HasBctPass) {
            if (Player.CurrentLocation != "The Bloodcrow Tavern") {
                  DisplayText "Looks like you're legit, kid.";
            }
            return true;
      }
      else {
            DisplayText "Scram kid. You're not wanted here.";
            return false;
      }
}
```

We don't need to check the current location in the false case since whenever it is false, we'll be leaving the location. Also should the player lose his pass while in the bar, we want him to be kicked out right away. Of course, since this is a rough place, we want to rough up the player a bit, and maybe have them lose their pass when they come in certain situations. The next subsection tells us how we might do that.

Entry Actions

Whenever a **`location`** is entered, all entry actions will be executed in the order they were added. Entry actions are void functions with no parameters. Entry actions will also be executed after a menu item's actions are executed if the **Refresh** property of that menu item is set to true. To add an entry action, we use the **+=** assignment operator:

`location_name``.`**`Entry`**` += `*`action_name;`*

The above statement will add the entry action *`action_name`* to the entry actions of *`location_name`*. You can also remove an entry action using the **–=** assignment operator instead of the **+=** assignment operator.

```
bcTavern.Entry += bcTavern_RoughUp;
bcTavern.Entry += bcTavern_ThiefPass;
function void bcTavern_Roughup() {
      if (Player.Level > Random(1,12) {
            DisplayText "A burly bar patron punches you…";
            DisplayText "You fall on some broken glass!";
            int damage = PlayerHp * Random(10,50) / 100;
            Player.Hp -= damage;
            DisplayText "You take " + damage + " damage!";
      }
}
function void bcTavern_ThiefPass() {
      if (Player.Class == 1) return; //Player is a thief.
      if (Random(1,112) > Player.Level + 100) {
            DisplayText "Suddenly your pass is missing!";
            DisplayText "You see a small thief scuttle away.";
            Player.HasBctPass = false;
```

```
        }
}
```

Now we have two entry functions. The first one will randomly cause a tavern patron to hurt the player, with low level players at higher risk, and level 12 or higher player completely safe. At a small chance, the second will randomly cause any non-thief player to have their tavern pass stolen, dependant upon player level again. An unlucky player might have both happen! Sometimes we'll want to also do something when a player exits the **location**. To find out how, read on.

Exit Actions

Whenever a **location** is exited, all exit actions will be executed in the order they were added. This occurs even if the player left due to a validity test, but only if the validity test failed after the player had entered the location. To add an exit action, we use the **+=** assignment operator:

location_name.**Exit** += *action_name;*

The above statement will add the exit action *action_name* to the exit actions of *location_name*. You can also remove an exit action using the **−=** assignment operator instead of the **+=** assignment operator.

Location Actions

Location actions execute repeatedly while waiting for a key press from the player. This is a good place to put actions such as checking for messages. . To add a location action, we use the **+=** assignment operator:

location_name.**Actions** += *action_name;*

The above statement will add the action *action_name* to the location actions of *location_name*. You can also remove an action using the **−=** assignment operator instead of the **+=** assignment operator.

Entering a Location

Now that we know all about creating locations, how to we use them in-game? Entering a location is as simple as using the location's **Enter** command:

location_name.**Enter;**

The above statement can be called from within any function. When a location is exited, program flow will return to the line after the **Enter** command.

Leaving a Location

To leave a location, simple use the **LeaveLocation** command from within any function. This signals the *NuklearScript* system to leave the most recent location as soon as menu command execution is complete.

Location Execution Order

Locations are executed in the following order:

1. Call to *location*.**Enter**
2. Do all validity tests.
3. If tests all passed, do allentry actions. Otherwise, return to next line after the *location*.**Enter** command.
4. While waiting for a keypress, do the following:
   a. Check if the LeaveLocation command was called. If so, do all exit actions and return to next line after the *location*.**Enter** command.
   b. Test validity. If failed, do all exit actions and return to next line after the *location*.**Enter** command.
   c. Do all location actions
5. Once a valid key is pressed, do the following:
   a. Run all the menu item's actions
6. Once a menu item's actions are complete, do the following:
   a. Check if the LeaveLocation command was called. If so, do all exit actions and return to next line after the *location*.**Enter** command.
   b. Check if the menu item's Refresh property is true. If true, do all entry actions and start again at step 4. If false, start again at step 4.

Game World Structure

The game world is composed of varying amounts of locations, functions, and commands, however to actual create a game world, we need the use of a special object: The **Game** object. This object has a few special properties. They are the **Game.MainMenu** location, and the **Game.EnterGame**, **Game.NewPlayer**, **Game.Maintenance**, and **Game.ExitGame** events.

Property	Type	Meaning
MainMenu	location	Special location that is the pre-entry menu for the game. Some **menuitem** from this **location** must call the **EnterGame** function.
EnterGame	Universal Function	Special function object that serves as the entry
NewPlayer	Universal Function	Special function object that acts to create a new player. This is called by the **EnterGame** function object whenever the player's **New** stat is **true**.
Maintenance	Universal Function	Special function object that is run once per game day, and is meant to perform any game day maintenance actions.
ExitGame	Universal Function	Special function that always executes as the last action of a player's stay. Calling this will result in logging out the player.

A Simple Sample Game

Let's explore the game object by creating a very simple sample game.

```
playerstat int Level;
playerstat int Hp;
playerstat int MaxHp;
playerstat int Strength;
playerstat int Defense;
playerstat int LastDayPlayed;
playerstat int BattlesLeft;
playerstat string Name;
playerstat bool Alive;
playerstat bool AtInn;

configuration normal int CurrentDay "Current Game Day" = 0;
configuration readonly int BattlesPerDay "Battles Per Day" = 25;

MainMenu.Name = "Main Menu";
MainMenu.SimpleMenu = mainmenu_menu;
MainMenu.Prompt = mainmenu_prompt;
```

```
MainMenu.Menu.Alignment = 0;

function string mainmenu_menu()
{
      string m;
      m = "Nuklear Test Game\n\nby Nuklear LORD Development Team\n\n” +
           “Version 1.0\n\n”;
      return m;
}

function string mainmenu_prompt()
{
      return "What say you, hero? : ";
}

menuitem mainmenu_E;
mainmenu_E.Key = 'V';
mainmenu_E.Text = "`2(`0E`2)nter The World";
mainmenu_E.Refresh = false;
mainmenu_E.Actions += Game.EnterGame;

menuitem mainmenu_Q;
mainmenu_E.Key = 'Q';
mainmenu_E.Text = "`2(`0Q`2)uit like a scared dog";
mainmenu_E.Refresh = false;
mainmenu_E.Actions += LeaveLocation; //Effectively exits the game
//We could have also called Game.ExitGame here as it checks to see if
//The player is online before actually running its functions.
```

That just about does it for the simple main menu. We could have also added some entry, exit, or location actions, though the possibilities are limited (general instructions, a game story… possibly some ads). You can also do a validity test, but you have even less options there (perhaps you only allow 50% of the players to play on any given day, and will block all further entry once that occurs.

```
function void MorePrompt()
{
      DisplayText "`2<`0MORE`2>";
      char temp;
      while (temp != '\r') {
            temp = GetKeyInput(false); //Gets a key, doesn’t echo char.
      }
}

Game.NewPlayer += SampleNewPlayer;

function void SampleNewPlayer()
{
      DisplayText "What ho, brave warrior!\n"; //Displays a string
```

```
	MorePrompt(); //Calls the MorePrompt Function, declared above.
	ClearScreen(); //Clears the screen.

	bool goodname = false;
	while (!goodname) {
		DisplayText "Pray tell me your name.\n";
		string name = GetTextInput(-1); // -1 = No input limit
		if (StringSize(CleanString(name)) > 20) {
			Display Text "\nGods above! Your name is too long!\n";
		}
		else {
			goodname = true;
			Player.Name = name;
		}
	}

	Player.Alive = true;
	Player.AtInn = false;
	Player.MaxHp = 20;
	Player.Hp = Player.MaxHp;
	Player.Strength = 15;
	Player.Defense = 5;
	Player.BattlesLeft = Config.BattlesPerDay;
	Player.LastDayPlayed = Config.CurrentDay;
	Player.Save;
}
```

Here we define a prompt used to break up text displays and requires the user to hit enter to continue, and then we add the **SampleNewPlayer** function to the **Game** object's **NewPlayer** event.

Inside the SampleNewPlayer function, we first display a greeting, followed by the prompt we made earlier, and then ask the player for a character name. We stick this is a while loop so we can check the length of the name and re-prompt for a new one if it is too long. Then we merely set a few initial stats and the function finishes.

StringSize is a built-in function that returns the number of characters in a string. **CleanString** is a built-in-function that returns a clean string, a string without any *NuklearCodes* in it (such as `2 for the green color, etc).

The **Game.NewPlayer** even is called from within the **Game.EnterGame** event; you cannot call **Game.NewPlayer** directly. Event though it is called from the **Game.EnterGame** even, it is often good to define the actions taken on a new player before figuring out what to do when any old player enters.

We call **Player.Save** at the end in order to ensure these stats are saved, as usually one of the first actions we add to **Game.EnterGame** is a **Player.Load**.

Note that we can add multiple functions to **Game.NewPlayer**, in the event that a module needs to have some actions performed when a new character is created.

```
Game.EnterGame += SampleEnterGame;

function void SampleEnterGame ()
{
      Player.Load;
      //First, we check if this player played today. If so, we do
      //Some beginning of day things.
      if (Player.LastDayPlayed < Config.CurrentDay) {
            if (Player.AtInn) {
                  //Split into two lines so it fits in this example:
                  DisplayText "You wake up cheerful and refreshed" +
                              "after your stay at the inn!";
            }
            else {
                  DisplayText "You wake up cheerful and refreshed!";
            }
            Player.Alive = true;
            Player.Hp = Player.MaxHp;
            Player.BattlesLeft = Config.BattlesPerDay;
            Player.LastDayPlayed = Config.CurrentDay;
      }
      //For all players entering the game, we do these things.
      CheckMessages() //declaration omitted for this example.
      //As game is developed, we might add more general entry
      //things here.

      //Lastly, we enter a location, the first real location
      //the player will be interacting will for this game.
      if (Player.AtInn) {
            Player.AtInn = false;
            Inn.Enter; //Enter the Inn location.
      }
      else { TownSquare.Enter; } //Enter the TownSquare location.
}
```

Here we define our game entry function. First, we load the player data so that it is fresh. They we do a comparison between the player's last day played and the current game day. Then if this is the player's first time in today, we then set all the 'beginning of new day' stats for that player. Afterwards, we do some general operations that are done for all players when they enter, regardless of whether it is their first time in today. Lastly we dump them into one of two locations based on the playerstat **`AtInn`**. Some games may always start a player out in the same players, or may have several (or hundreds!) of possible starting locations, depending on how you want to program your game.

Note that **`CheckMessages`** is not a built-in function. We would likely develop some sort of character to character mailing system in our game, and this is meant to represent such a function.

Note that we can add multiple functions to **`Game.EnterGame`**, in the event that a module needs to have some actions performed when a player enters the game world.

```
function bool CheckDeath()
{
	if (Player.Hp <= 0 || !Player.Alive) {
		Player.Alive = false;
		Player.Save;
		Game.ExitGame;
		return false;
	}
	return true;
}
```

Here we have what will probably be the most common kind of validity test in any game. If we add this as a validity test to all our locations, we ensure that a dead player doesn't get to remain living. We save the player's stats for good measure as well. We return false as a matter of course in the dead condition, but it will never actually be called since **Game.ExitGame** will always end the game session.

Note that you can call Game.ExitGame at any time and the system will properly handle it. Thus you will want to craft your game exit functions to ensure that all data is properly saved.

```
Game.ExitGame += SampleExitGame;

function void SampleExitGame ()
{
	//We might also clear any temporary stats here, or
	//store statistics about the game, etc.

	if (Player.Hp <= 0 || !Player.Alive) {
		Player.Alive = false;
		Player.Save;
	}
}
```

Our exit game function looks almost the same as our **CheckDeath** function above, however it is important to note that **Game.ExitGame** will always be called in the event of a disconnection or if you reach the end of the **Game.EnterGame** function and so it is the last bastion of defense against unwanted exit conditions.

You would now, at this point in time, craft all your locations and any other stats and functions you need.

Game Flow Order of Execution

Game Flow follows this order of execution:

1. Client connects to the server and sends login information.
2. Player object is initialized with empty data.
3. If login name is in the database, player **ID** is set and the **New** player stat is set to **false**. Otherwise, the **New** player stat is set to **true**.

4. **Game.MainMenu.Enter** is called.

5. System waits for **Game.EnterGame** to be called.

6. As the very first action in **Game.EnterGame**, if **Player.New** is true, **Game.NewPlayer** is called and all **NewPlayer** functions are executed in the order they were added.

7. After **Game.NewPlayer** excutes (if needed), all **EnterGame** functions are executed in the order they were added.

8. When the last **EnterGame** function ends, **Game.ExitGame** is called. All **ExitGame** functions are executed in the order they were added. **Game.ExitGame** may be called at any time. If a client disconnects, **Game.ExitGame** will automatically be called.

Maintenance Actions

Use the Game. Maintenance event to add functions that you want to run once every game day. The **Game.Maintenance** event is called at the time configured in the configuration application. Keep in mind that there is no **Player** object while running these functions. You can, however, use the **Enemy** object to load data from players. Any **location.Enter** commands will be ignored by the maintenance, as will any input commands.

```
Game.Maintenance += SampleMaintenance;

function void SampleMaintenance ()
{
	//Things that must be done once per game day go here.
}
```

Gateways and Random Selections

Two special structures exist to make adding modules easier to add to the system. They are the gateway, which is a special kind of menu, and the random selection, which will randomly select a function from its list of functions.

Gateways

Gateways are almost identical to locations. However they only have the following properties:

Property	Type	Meaning
Name	string	Name of the **gateway**.
Prompt	string	Prompt generated for this **gateway**.
Header	Universal string Function	String containing the menu for this **gateway**. Displayed only if **HeaderFile** is an empty string.
HeaderFile	string	File containing the header for this **gateway**
HeaderSection	string	Subsection of the **HeaderFile** that contains the menu for this **gateway**.
Tests	Universal Test Function	Contains a list Universal Testing Functions that determine if the **gateway** should be entered.
Entry	Universal Function	Contains a list of Universal Functions that are executed upon entry into this **gateway**.
Exit	Universal Function	Contains a list of Universal Functions that are executed upon leaving this **gateway**.
Actions	Universal Function	Contains a list of Universal Functions that are executed continuously while waiting for a key press in this **gateway**.
Gates	Universal Function	Functions to be added as menu items.
KeyColor	char	Character representing the color to use for the key indicators of the gateway menu.
BubbleColor	char	Character representing the color to use for the parentheses around the key indicators of the gateway menu.

To declare a gateway, use the following as a top level code statement:

```
gateway gateway_name;
```

The **Header** Property behaves the same as the **SimpleMenu** property of a location. **HeaderFile** and **HeaderSection** are similar to the **MenuFile** and **MenuSection** properties of a location, however they should not include the menu item descriptions, merely the header displayed before the options.

Gateways are meant to be used as hubs to travel between regions in a game, or to allow for one place when extra modules are built off of. As such, when you add functions to the **Gates**, they are numbered from 0 to 9, then A to Z (Attempting to add more than 36 will result in an error).

To add a gate to the **gateway**, use the following command, which should be done at the top level of code:

```
gateway_name.Gates.Add(function_name, string_description);
```

The string description the name you wish to appear when the gateway menu is displayed.

To remove a gate from the **gateway**, use its remove command:

```
gateway_name.Gates.Remove(function_name);
```

This will remove the function from its list of gates.

As an example of a gateway in action, let's say a player has made three add-on regions to his game as new modules. He wants them all to be access by a gateway he called, "Other Regions." He might build the **gateway** like this:

```
gateway otherRegions;
otherRegions.Name = "** OTHER REGIONS **";
otherRegions.Header = otherRegions _header;
otherRegions.Prompt = otherRegions _prompt;
otherRegions.Tests += CheckDeath;
otherRegions.Actions += CheckMessages;
otherRegions.KeyColor = '%';
otherRegions.BubbleColor = '9';

function string otherRegions _header()
{
     return "`%" + otherRegions.Name + "\n\n"

function string otherRegions _prompt()
{
     return "`2Where shall you venture, `0" + Player.Name + "12? : ";
}
```

```
//Somewhere in module, CavernOfDeath.ns:
otherRegions.Gates.Add(cavernEntry, "`7The Cavern Of Death");

// Somewhere in module, BloodcrowForest.ns:
otherRegions.Gates.Add(bcForestEntry, "`4Bloodcrow `2Forest");

// Somewhere in module, Valhalla.ns:
otherRegions.Gates.Add(valEntry,
                       "`1Val`9hal`!la, R`%ealm 1!of t`9he G`1ods");
```

As a result, his Gateway might look like this:

```
** OTHER REGIONS **

(0) The Cavern of Death
(1) Bloodcrow Forest
(2) Valhalla, Realm of the Gods

Where shall you venture, Gandalf? :
```

Random Selection Structures

Random selection structures are used to allow for easy modularity of special events. Functions are added to them, and then when the **randomselection** is called, it will randomly choose one of its functions to execute, based on the frequency property of that function.

To declare a gateway, use the following as a top level code statement:

```
randomselection randomselection_name;
```

To add a function to the **randomselection**, use the following command, which should be done at the top level of code:

```
randomselection_name.Add(function_name, int_frequency);
```

The frequency is the relative frequency of the function compared to the other functions. A function with a frequency twice the value of another function's frequency will be called twice as often on average. For example, if you had six functions added to your random selection and four of them had a frequency of 2, and the other two had a frequency of 1, the ones with a frequency of 2 would be called 20% of the time, and the others would be called 10% of the time. To put it another way, the total frequency in this case adds up to 10, so the first four would be called 2 in 10 times, and the others 1 in 10 times.