Types

At a Glance

```
// Commonly Used
int a = ;       // integer
float b = 3.14f;    // floating point

bool c = true;       // Boolean (true/false)
bool d = false;

string e = "Amnesia";  // string

////////////////////////////////////////////////////////////
// Signed Integer Types
int8 f = -2;        // -128 to 127
int16 g = 5;         // -32,768 to 32,767
int h = 100;         // -2,147,483,648 to 2,147,483,647
int64 i = 8;         // -9,223,372,036,854,775,808 to
9,223,372,036,854,775,807

// Unsigned Integer Types
uint8 j = 255;       // 0 to 255
uint16 k = 10;       // 0 to 65,535
uint l = 1234;        // 0 to 4,294,967,295
uint64 m = 56;       // 0 to 18,446,744,073,709,551,615

// Script engine is optimized for 32-bit data types
// int is alias for int32
// uint is alias for uint32


// Floating Point Types
float n = 1.234f;
double o = 5.67;

float p = 10.0e-5f;     // exponent notation
double q = 6.022141e+23;    // exponent notation


// Classes
MyClass obj1;       // instantiates the obj object to default value
MyClass obj2();       // likewise, explicitly invoking the constructor
MyClass obj3(5);     // constructor with a parameter

// Classes - Using Constructors As Functions
// NOTE: Not allowed in global scope!
MyClass obj4 = MyClass();    // default constructor
MyClass obj5 = MyClass(10); // constructor with a parameter

MyClass@ ptrObj1;      // A null handle (pointer) to an object of type
MyClass
MyClass@ ptrObj2 = @obj2; // ptrObj2 holds a reference to obj2

// Arrays
int[] arrayA(5);     // A 5-element integer array (uninitialized)
int[] arrayB(5, 3);        // A 5-element array, all elements set to 3
uint[] arrayC = { 1, 1, 2, 3, 5, 8 };    // explicit initialization
uint[] arrayD = { , , 5, 5, , }; // 6-element array, with some elements
initialized


MyClass[] objArray(10);   // An array of objects
MyClass@[] ptrArray(10);  // An array of object handles (pointers)

// Function Handles (Function Pointers)
MyFuncPtr@ funcPtr = @MyFunc;
```

Discussion

Data types define what kind of data a variable can store, and what sort of operations can be applied to that data. For Amnesia scripting, for the most part you'll be using only four data types, but you might occasionally want to take advantage of some of the others. The four common data types are:

- **int** - which represents integer numbers,
- **float** - which is used to represent real numbers in the so-called floating point format (it just refers to the internal representation the computer uses),
- **bool** - used to represent Boolean values (truth values); can represent only two values, either `true` or `false`,
- **string** - used to represent character strings, i.e. - text.

The `int` type is an alias for the `int32` type; this means that each int variable uses 32 bits, or 4 bytes, of memory to store an integer number. Since the amount of memory *is finite*, not all possible numbers can be represented in a computer, but only numbers in a certain range - however, this range is usually more than enough.
For the int type, the range is: `-2,147,483,648` to `2,147,483,647`.

The `int` type is a *signed* type, which means that it can be used to represent both positive and negative numbers. Each of the signed integral types has an *unsigned* variant. The unsigned counterpart of int is the `uint` type.
Unsigned integers are non-negative (zero or positive) by definition; this means that the in-memory representations which were previously needed to represent negative numbers are now free, so the range can be pushed further in the positive direction. Thus, `uint` can represent bigger numbers than `int`, but both `uint` and `int` can take on the same amount of different values. The range of the `uint` type is: to `4,294,967,295`.

Other Integral Types

The script language also provides other integral types, which can take up less or more memory. The more memory a type consumes, the greater number of different values it can represent. However, the script engine is optimized for 32-bit data types, so using the types of other sizes should be reserved for special circumstances.

The other integer types are `int8`, `int16`, `int64`, `uint8`, `uint16` and `uint64`. The suffix indicates how many bits each of them takes up. Their ranges are listed in the “At a Glance” section above.

Real Numbers - Floating Point Types

There are two floating point types available. One is the already mentioned `float` type, and the other is `double`.
The `float` type takes up 32 bits (4 bytes) of memory, and it can represent, with varying precision, real numbers in the range +/- 3.402823466e+38. If you're not familiar with the exponent notation, the e+38 in the end is a shorthand for “* 10^38”, which means that in order to get the actual values at the extreme positive and negative ends of the range, the number 3.402823466 should be multiplied with a really, really big number, 1 followed by 38 zeros! So, `float` can represent both really small and really big numbers, but the bigger the number the less precise is the representation in terms of decimal places right of the dot (this is why the format is called “floating point”).

Again, because there are infinitely many real numbers, and the computer can only use a finite amount of memory to represent them, not all real numbers can be exactly represented by both of the floating point types. But, this is generally not a problem - for Amnesia scripting, you almost never need to worry about this. Here are some characteristics of the `float` type:

- *Range:*`+/- 3.402823466e+38` (or approximately: +/- 3<*38_zeros_here*>.0)
- *Smallest positive value:*`1.175494351e-38` (or: 0.<*37_zeros_here*>1175494351)
- *Precision:*`6-7 digits`

The `double` type is also a floating point type, but it takes up 64 bits (8 bytes) of memory, double the amount the `float` type uses (thus the name). It can represent a wider range of values, and with a greater precision.

- *Range:*`+/- 1.7976931348623158e+308` (or approximately: +/- 1<*308_zeros_here*>.0)
- *Smallest positive value:*`2.2250738585072014e-308` (or: 0.<*307_zeros_here*>22250738585072014)
- *Precision:*`15-16 digits`

Real number literals are considered to be of type `double` by default. When assigning values to floating-point variables, you can append the 'f' suffix to numerical literals to indicate that they should be treated as `float`-s instead, and avoid unnecessary conversion, like this:
```
float number = 9.81f;
```

Boolean Type

The `bool` type is used to represent the values `true` and `false`. Variables of `bool` type are useful *as*

indicators. You can use them to represent binary states (for example, on/off switches), or indicate if some event happened or not. You can use them as return values of functions, to indicate the success or failure of an operation.
The `bool` type is named after George Boole, an English mathematician, philosopher and logician, who first defined an algebraic system of logic in the mid 19th century.

For more info about the operations available for the `bool` type, see Comparison Operators and Using Logical Operators.

Strings

The `string` type is not native to AngelScript, but is exposed by the HPL2 engine instead, and it is used to represent textual data. It supports string concatenation via the + operator. To *convert other types* to their textual representation, simply append them to an empty string, like this:

```
int anInteger = 10;
string aString = "" + anInteger;  // "10" is stored in aString variable
```

Classes

You'll learn more about classes in the Classes section (to do…). Here it will suffice to say that classes provide a way for the scripter to define custom, more complex types, which can be used to represent various concepts, like states, game entities, or behaviors. For example, you can write a class to represent a pass-code lock mechanism to be used in a map of your custom story. This class would define the possible states (values) for the lock mechanism, as well as operations that are available/allowed on it. Once defined, the class, which is actually a newly defined type, can be used to create variables, in the same way as any other type you encountered so far. It would be very simple to declare several variables of the “pass-code lock”-type to represent several locking mechanisms that might exist in your map, and yet, all of them would use the same peace of code to do their job, but would maintain separate state variables. This may all sound complicated, but it really isn't, and actually helps simplify the script, and avoids code duplication (which is inherently error prone).
The “At a Glance” section at the top summarizes how classes are used to declare variables (known as *class instances*, or *objects*).

Object Handles (Object Pointers)

While you can think of variables of other data types as of containing their value directly, object handles (or objects pointers) are a special kind of data type which stores an indirect reference, or a pointer, to the memory location which holds the actual value (object). This enables you to have two or more object handle variables pointing to the same value; using and making changes through any one of them will be reflected on all the others. This is useful in some non-basic scenarios, so it is explained in another place (to do…).

Arrays

Arrays are sequential collections (or lists) of data of the same type. Basically, declaring an array of *n* elements is in a way similar to declaring *n* variables of the same type at once. To declare an array of a certain type, append `[]` to the name of that type. For example, an array of integers is denoted like this:
`int[]`.

For HPL2 scripting, you must specify the size, or length, of the array (that is, the number of elements it can hold) when you declare it, as this cannot be changed later on. You do this by placing the size in parenthesis, and appending that to the name of the array variable:
`int[] anArray(100);` ← declares an int array containing 100 elements

When declared like this, all elements are left uninitialized, and contain whatever junk data was at the memory locations assigned to them, so be careful to initialize the elements before you use them. You can also initialize the elements when declaring the array, in a number of ways.

You can initialize all the elements to the same value by passing a second parameter to the array's constructor:
`int[] anArray(100, 0);` ← all elements initialized to 0

You can also use a different notation to explicitely initialize each of the elements:
`int[] anArray = { 128, 64, 32, 16 };` ← an array of size 4, with initialized elements

Using the same notation, you can leave some of the elements uninitialized:
`int[] anArray = { , , 32, };` ← an array of size 4, with only the 3rd element initialized to 32

Each of the array elements is accesed by it's index, which corresponds to the element's position in the array. However, remember that indices start from 0, not from 1, so an array of length 4 will have it's elements at indices (locations) 0, 1, 2, and 3; that is, *length-1*.

You can visualize an array like this:

```
int[] anArray = { 7, 2, , 8 };  // an array of size 4, with initialized elements

// Visualization:
// Array values:   [7][2][0][8]
// Indices:         0  1  2  3       <-- size 4
```

You'll learn more about arrays in the Array section of the guide.

Function Handles (Function Pointers)

Function handles are user-defined types which enable you to represent *a script function as a variable*. This can be useful if you want to change the way the script behaves dynamically, during gameplay. As function handles are variables, they can be passed as parameters to other functions. As this is a somewhat advanced topic, function handles are explained in a separate section of this guide (to do…). You can also learn more about them here. The “At a Glance” section on this page demonstrates how to declare and initialize a function pointer for easy reference.