GSC Language Reference

Keywords

- class
- function
- var
- return
- wait
- thread
- undefined
- self
- world
- classes
- level
- game
- anim
- if
- else
- do
- while
- for
- foreach
- in
- new
- waittill
- waittillmatch
- waittillframeend
- switch
- case
- default
- break
- continue
- false
- true

notify
endon
assert
assertmsg
constructor
destructor
autoexec
private
const
isdefined
vectorscale
gettime
waitrealtime
profilestart
profilestop
#using_animtree
#animtree
#using
#namespace
#precache
.size

Operators

||
&&
|
&
^
==
===
!=
!==
<
>
<=
>=
<<
>>
+
-
*

/
%
!
~
=
->
++
--
|=
^=
&=
<<=
>>=
+=
-=
*=
/=
%=

Special Tokens

$
#
.
//
/*
*/
/#
#/
/@
@/
[[
]]
[
]
(
)
0x

File types

.gsc

This file represents a chunk of server side script containing one or more namespaces.

This is the assumed extension for resolving #using declarations when the source file has a .gsc extension.

.csc

This file represents a chunk of client side script containing one or more namespaces.

This is the assumed extension for resolving #using declarations when the source files has a .csc extension.

.gsh

This file contains injectable code or macros. This file extension is recognized by convention, but no formal requirement exists enforcing the extension.

Code is inserted into a gsc/csc using #insert. The code is injected directly into the source by the preprocessor and is parsed as if the lines were present in the gsc/csc that it is inserted into.

Preprocessor

A simple preprocessor is supported by the compiler. The preprocessor resolves all macros and keywords in the first compiler pass. The compiler then generates code in the second pass.

#define NAME VALUE

#define NAME(arg) VALUE

This command defines a substitution macro and can take one of two forms: Either as a simple replacement value or with a list of arguments that can be used in the macro. Substitution macros are processed and replaced at the target location. Then, the preprocessor restarts the parse again at the beginning of the replaced text. This means that you can nest macros and they will properly resolve. Be careful to avoid recursion within a macro that doesn't read a terminal state.

NAME is the keyword that the preprocessor will locate and substitute in the source with ***VALUE***.

NAME is case sensitive.

VALUE can contain any text.

A macro can span multiple lines by using \ as the last character on the line.

Examples:

```
#define PITCH              0
#define YAW                1
#define ROLL               2

#define PI                 3.14159
#define RGB(__r,__g,__b)   (__r/255,__g/255,__b/255)
#define RED                ( 1, 0, 0 )
```

#insert FILEPATHNAME;

This command inserts another file, ***FILEPATHNAME,*** at this point in the source. By convention .gsh files are used, but this rule is not enforced and any file name or extension may be used. All file paths are rooted at "%TA_GAME_PATH%/share/raw". All of the content of the inserted file will appear at the line number the #insert is on. This means that error output from the game will identify line numbers correctly with reference to the original document. But, any error generated from the entire file inserted will be reported at the single line where the insert is located. The insert command can legally appear anywhere in the file, since it is handled by the preprocessor.

Example:

```
#insert scripts\shared\shared.gsh;
```

#if CONDITIONAL

#elif CONDITIONAL

#else

#endif

The preprocessor will evaluate the ***CONDITIONAL*** expression and remove blocks of code from compilation. It does not alter the line numbers, so the compiler will report errors correctly based on the original source file.

Some macros are automatically provided by the compiler:

- __FILE__ this automatic macro is substituted with the name of the file the compiler is currently processing.
- __LINE__ this automatic macro is substituted with the line number where the __LINE__ macro is present.
- FASTFILE this automatic macro is substituted with the name of the fastfile the compiler is currently generating.

Example:

```
#if XFILE_VERSION >= 553
              extracam_data.useHeadIndex = GetFirstHeadOfGender( GetHeroGender(
       characterIndex, sessionMode ), sessionMode );
#endif // #if XFILE_VERSION >= 553
```

Syntactic Structure

A GSC/CSC program requires a program to be constructed as two basic blocks: A set of "#using" instructions, and a set functional blocks.

```
#using foo;

#using bar;


function baz()

{

}
```

It is a syntax error to include a "#using" statement after the first function is declared.

#using FILEPATHNAME;

The using command informs the compiler that an additional file should be scanned to locate unresolved references. These references might be function calls or class references. The referenced file is incorporated into the active script set by the game at runtime. Any file referenced with a #using will therefore have an opportunity to execute script, even if no direct references to any exported functionality are present.

FILEPATHNAME is the file to be included for resolving references. All file paths are rooted at "%TA_GAME_PATH%/share/raw".

function [OPTIONS] IDENTIFIER([ARGUMENT[=VALUE]][,ARGUMENT]*)

Function declarations declare discreet blocks of code that may or may not return a value to the caller. By default, all functions are assumed to be exported and available to external callers. However, this behavior can be modified. All functions have a built in variable, ***self***. This variable is a reference to the variable passed in through the method calling notation. If the call was made without that format, then the ***self*** object is the same as the ***self*** object of the caller. The default ***self*** object is the global reference ***level***.

OPTIONS can provide one or more alterations to the way a function can be used.

- ***private*** declares a function that is restricted to the current file. External callers outside of the context of the current file cannot resolve this function.
- ***autoexec*** declares a function that will be run automatically by the game just before the level's main entry point is executed. Note that with this keyword it is possible to execute script that is not explicitly referenced, but is included via a ***#using*** statement. The order of execution of ***autoexec*** functions is undefined and should not be relied upon. If an explicit order of execution is required it is up to the implementer to place appropriate conditionals and flow control in the implementation of the function.

IDENTIFIER is the unique name in the current namespace. You can have identically named functions as long as the namespace differs.

A function can optionally contain one or more ***ARGUMENT*** values separated by commas. Each ***ARGUMENT*** can also optionally contain a default value declaration. The default value must resolve to a value. It cannot be a function reference or require further resolution. If a caller passes ***undefined*** into a function that declares a default value for an ***ARGUMENT*** then the default value is substituted instead. Any argument can declare a default value. There is no requirement, unlike C for example, that requires all following arguments to also declare default values.

An ***array*** type ***ARGUMENT*** can optionally use the ***&*** operator (e.g. ***&ARGUMENT***) to declare them as pass by reference. The default behavior for callers is to pass arrays by copy. This is potentially much slower and requires greater resource usage. If passed by reference, the original copy of the array variable is operated on directly by the called function.

class IDENTIFIER [: PARENT]

this statement creates a ***class*** with zero or more ***class*** variables, methods, and an optional constructor or destructor. Single inheritance is supported by specifying a ***PARENT*** class. A ***class*** declares a block in which all ***class*** variables and methods are present. The ***self*** variable in all ***class*** functions is a reference to the class itself. It is an error to make a direct method style call on a ***class*** function.

The constructor is automatically called on object creation and the destructor is called when the last reference to the object is released. It is not possible to directly call these functions.

Within a ***class*** object, ***class*** variables may be declared by using the ***var*** keyword. These variables are available via the ***self*** reference and are inherited by any child class. However, use of self is optional and the variable may simply be referenced directly from within any ***class*** function. Be aware of name collisions and scoping rules, however.

Example:

```
class cTeamGather
{
```

```
        var e_gameobject;

        var    n_font_scale;
        var    v_font_color;


        constructor()
        {
               e_gameobject = undefined;
               n_font_scale = 2.0;
               v_font_color = ( 1.0, 1.0, 1.0 );
        }

        destructor()
        {
        }
}
```

#namespace NAMESPACE;

This statement changes the current namespace state to ***NAMESPACE***. This state persists until the file ends or another ***#namespace*** statement is encountered. Any function declared will belong to the current ***#namepace*** state. If no ***#namespace*** is specifically declared, then the name of the file is the current namespace.

A class declaration alters the current namespace state to match the class name within the class definition block. The previous namespace is restored at the end of the class definition block.

Function calls made that do not use an explicit namespace are assumed to use the current namespace state. If the compiler fails to find a function match in that namespace it will assume the namespace is ***sys*** (the default builtin function library supported by the game). Builtin functions can always be fully qualified by using ***sys.***

Example (in the file sound_shared.gsc):

```
#using scripts\shared\util_shared;
#insert scripts\shared\shared.gsh;

#namespace sound;
function foo( alias, origin = (0,0,0) , ender )
{
}

#namespace audio;
Function bar()
{
}
```

This will create a function sound::foo() that takes three parameters, and supports a default value for the 2nd parameter. It also creates audio::bar() which takes no parameters. Within the namespace sound the use of sound:: in a call is optional for functions within that namespace.

#precache(TYPE, VALUES);

This statement identifies assets that the script references and should be included the linker in the final fastfile.

TYPE identifies the type of asset. Valid types are:

"vehicle"
"model"
"playercharacter"
"aitype"
"character"
"xmodelalias"
"weapon"
"zbarrier"
"rumble"
"shellshock"
"xcam"
"destructible"
"streamerhint"
"headicon"
"statusicon"
"locationselector"
"menu"
"material"
"string"
"debugstring"
"eventstring"
"triggerstring"
"objective"
"fx"
"lui_menu"
"lui_menu_data"
"client_fx"
"client_tagfxset"

Keyword Reference

Class – declares a class, its associated variables and functionality.
Function – declares a function.
Var – valid within a class declaration. Declares a class variable accessible by all class functions.
Return - returns a value from a function. Any function that does not explicity return a value is assumed to return undefined.
Wait – causes the current execution context to go into a sleeping state immediately for the specified time. The next pending thread of execution will begin immediately.
Thread – creates a function call that has a separate thread of execution. This call can optionally pause its state of execution prior to returning. In that case the caller is immediately resumed at the point following the threaded call.
Undefined – a special value that indicates no value is present. An array can contain undefined values without collapsing the array automatically.
Self – a special value that represents the variable used in a method type call.
World – a persistent global variable (struct) whose contents survive from level to level. Note that there are restrictions on the data types that may be placed into the world object. Lifespan is the duration of the game execution.
Level – a global variable (struct) for general use during the execution of a level. Lifespan in the duration of a level.
Game – a global variable (array) for general use during the execution of a level. Lifespan is the duration of a match.
If – if conditional clause construct.
Else – if condition clause construct.
Do – do while loop construct
While – do while loop construct or while loop construct.
For – for loop construct.
Foreach – foreach loop construct.
In – foreach loop construct.
New – create a new instance of a class object.
Waittill – wait for a specific notify to occur.
Waittillmatch – wait for multiple specific notifies.
Waittillframeend – move the current thread of execution to the end of the pending list.
Switch – switch state construct.
Case – switch state construct.
Default – switch state construct.
Break – break the current execution and resume at the tail of the current block.
Continue – jump to the conditional clause in a loop.
False – Boolean false.
True – Boolean true.
Notify – send a notify signal that can get intercepted by all pending threads at a waittill.

Endon – terminate the current thread of execution immediately on receipt of the specified signal.
Assert – Only tested when devblocks are enabled. Break execution if the supplied condition evaluates as false.
Assertmsg – same as assert, but provide an output string to print if the supplied condition evaluates as false.
Constructor – specifies the constructor function for a class. Does not support parameters at this time.
Destructor – specifies the destructor function for a class. Called automatically when final reference to a class object is released.
Autoexec – option for a function. Function is executed automatically prior to the level entry point getting called.
Private – removes a function from the export table. No external calls are allowed.
Const – declare a variable as fixed and unchanging. Helps the compiler produce more optimal code by allowing some compile time optimizations to occur.
Isdefined – returns true if the variable contains a value other than undefined.
.size – evaluates the expression as the number of elements in an array object.

Operator Reference

|| - Boolean or
&& - Boolean and
| - logical or
& - logical and
^ - logical xor
== - evaluate equality
=== - super equality. True only if both sides are same type and same value.
!= - evaluate inequality.
!== - super inequality. True only if both sides are different types or same type and different values.
< - less than.

> - greater than

<= - less than or equal to
>= - greater than or equal to
<< - shift left
>> - shift right
+ - addition
- -subtraction
* - multiplication
/ - division
% - modulus

! – Boolean not
~ - logical not
= - assignment operator

-> - call class method

++ - post increment

-- - post decrement

|= -logical or with assignment
^= -logical xor with assignment
&= - logical and with assignment
<<= - shift left with assignment
>>= -shift right with assignment
+= - addition with assignment
-= - subtraction with assignment
*= - multiplication with assignment
/= - division with assignment
%= - modulus with assignment

Special Tokens

#”” – compile time hash of string
. – reference sub variable in structure
// - line comment
/* - begin block comment
*/ - end block comment
/# - begin dev block
#/ - end dev block
/@ -begin documentation block
@/ - end documentation block
[[- begin expression evaluation
]] – end expression evaluation
[- begin array reference
] – end array reference
(- begin parameter list
) – end parameter list
0x – indicates following number is base 16
… - when used at end of function parameter list, indicates variable argument list. Additional arguments appear as a local array variable named ***vararg.***

Examples

```
#using scripts\codescripts\struct;

#insert scripts\shared\shared.gsh;

#namespace foo;

#define BAR    0
#define BAZ(_x) _x

#precache( "string", "TEAM_GATHER_TEAM_STEALTH_ENTER" );


class Boo
{
var far;
        constructor()
        {
                far = 1;
        }

        destructor()
        {
        }

        function faz( value = 0 )
        {
                far = value;
        }
}

//inheritence
class Faz : Boo
{
        var far2;
        constructor()
        {
        far2=2;
        }

        function faz( value1 = 1, value2 = 2 )
        {
                Boo::faz(value1);
                Far2=value2;
        }
}

function flop()
{
        //class syntax
        boo_object = new Boo();
        faz_object = new Faz();
        [[boo_object]]->faz();
        [[faz_object]]->faz(undefined, 1);

        //all of these loops are functionally identical
        A = [];
        I = 0;
```

```
        Do
        {
                A[i] = I;
                I++;
        }
        while (I < 10);

        A=[];
        for (i=0;i<10;i++)
        {
                A[i] = I;
        }

        A=[];
        while (i<10)
        {
                A[i]=I;
                I++;
        }

        //foreach syntax
        foreach(key,value in a)
        {
                Println( “key is “ + key + “ and value is “ + value  + “\n”);
        }

        Println(“bar = “ + BAR);
        Println(“baz = “ + BAZ(10));


        v = 1;
        v2 = “default”;

        switch(v)
        {
        case 0:
                V2 = “0”;
                Break;
        case 1:
                V2 = “1”;
                Break;
        default:
                V2 = “default”;
                Break;
        }
}
```