cDebug 1.15.4

[Introduction](#)
Maps or no maps
A teaser
Parameters
    Location
    Names
    Value arguments
    Square brackets in $names
Two simple examples
Arrays
    Array elements that are arrays
    Slices of arrays
Maps
    Filtering using a regular expression
Multi-level filtering
    The same sub-expression for an array and a map
    Sub-expression with negative numbers
DLL structures
    Coding a tag
    Without coding a tag
    So why not always specify a tag?
    Limitations of detection
    What if the detected structure does not match what you specified?
    When you specify an incorrect tag
    Overriding a specified tag
    Padding
    Pseudo-code of per-call and detected reporting
Easy coding
Copying location and data to the Clipboard
Conditional debugging
Changing the number of elements displayed temporarily
Customizing cDebug permanently
When you are finished debugging
Error detection
Naming of functions and global constants and variables
Files
Caveats
Known limitations
Edit history
Acknowledgements

Introduction

The cDebug script includes four main debugging UDFs: `_GuiDebug()`, `_ConsDebug()`, `_ClipDebug()` and _FormatValsForDebug(). They all dump the values of all AutoIt subtypes and expressions, in a structured manner, including, nested arrays and maps, and *slices* of them. They handle arrays with up to 3 dimensions.

For DLL structures, if you specify a tag, cDebug checks for differences between it and what it detects. If you only specify a structure variable, it reports the structure it detects, with the values of elements.

It does much more than MsgBox(), ConsoleWrite() and _ArrayDisplay(), in a definitely user-friendly manner, and does its best to avoid hiding your code in SciTE.

It is an alternative to a graphical debugger, offering GUI output. The first three UDFs, as indicated by their names, dump to a GUI, to the SciTE console, and to the Clipboard respectively.

While `_FormatValsForDebug()` is called by the first three main UDFs, it is useful on its own, when called by user functions. It returns the data as a string, with lines delimited by `@CRLFs`.

There is a fifth main function: `_MsgDebug()`. In earlier versions, it dumped to a message box, but, it did not handle many lines of dump gracefully. It now does the same as `_GuiDebug()`, and is retained to avoid breaking scripts. It is intended for use with `_ConsDebug()`.

Because their parameters are the same, and these UDFs only differ as to destination, one description is given for all three. The examples are for `_GuiDebug()`. Simply replacing the beginning of this function name with `_ConsDebug()` will send the same information to the Console.

There is also another UDF `_ChangeElementLimitForDebug()`, that you may not need. It affects how much is reported. It takes one argument: the new element limit.

cDebug.au3 includes an example function, `_cDebug_Example()`, which is at the end of the script. It takes no parameters. It shows many of the capabilities of the main UDFs. My experience, over some years, is that rarely does one use so many of the capabilities together, but the example shows it can be done – and what is available. Examine its code, try it, and see!

Maps or no maps

CDebug.au3 was written when maps were an experimental feature: while the help for AutoIt 3.3.15.0 does not say *use maps at your own risk*, this is/was the situation. So, at least until the map feature is completely reliable, there are two streams: `cDebug.au3` and `cDebug no maps.au3` . They are identical, except in the latter code requiring maps is commented out. It has been tested on AutoIt 3.3.14.5.

If you are running AutoIt 3.3.15.0 or a later beta version of AutoIt that supports maps, you must insert `#AutoIt3Wrapper_Version=B` at the top of your script and `#include cDebug.au3`; otherwise just #include `cDebug no maps.au3` .

While the AutoIt developers have discovered bugs in handling maps in AutoIt, none have been encountered in developing cDebug.

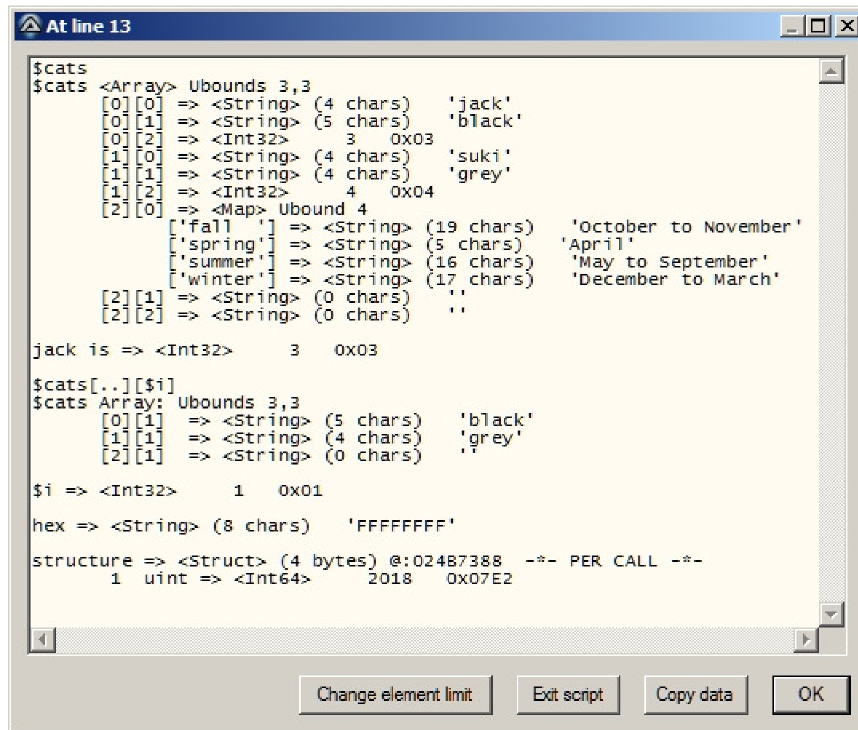
This document assumes that maps are part of AutoIt. If you are including `cDebug no maps.au3`, ignore maps in this document.

A teaser

This script:

```
#AutoIt3Wrapper_Version=B       ; beta 3.3.15.0 or greater is mandatory
#include "cDebug.au3"
Local $seasons[]
$seasons.summer = 'May to September'
$seasons.spring = 'April'
$seasons.fall = 'October to November'
$seasons.winter = 'December to March'
Local $aCats[3][3] = [['jack','black',3],['suki','grey',4],[$seasons,'','']]
Local $i = 1
Local $tStruct = DllStructCreate('uint')
DllStructSetData($tStruct,1,2018)
_GuiDebug('At line '&@ScriptLineNumber,'$cats,jack is,$cats[..][$i],$i,hex,structure{uint}', _
$aCats,$aCats[0][2],$aCats,$i,Hex(-$i),$tstruct)
```

produces



The GUI is dynamically sized to fit the data. The results shown below were inserted into this document after copying the data to the Clipboard by clicking on the **Copy data** button. To enhance the readability of the results, the title of the GUI, which is the first line on the Clipboard, has been omitted from all but the first few examples.

The same results would be produced if _ConsDebug() was called, rather than GuiDebug(). To change from _GuiDebug() to _ConsDebug() just replace Gui with Cons. It couldn't be easier!

The **Change element limit** button is explained below. If you choose a new limit, the display is refreshed using the new value.

Parameters

The required parameters are, for all the UDFs: $location, $name, and $val1. They can be followed by $val2 thru $val19, which are optional. $val1 through $val19 are called *value parameters*.

Location This parameter is used to show where in the user code one of the main UDFs is called. A typical argument is @ScriptLineNumber but, *except for one special character*, it can be any number or string. In a dump, the value of $location is:

- for _GuiDebug(): the title of the message box
- for _ConsDebug() and _ClipDebug(): the first line of the dump to the console or Clipboard.
- for the Copy Data button, the first line that is placed on the Clipboard

The special character is ^ : for _GuiDebug() and _ConsDebug(), it causes dumping to the Clipboard as well as to a GUI or the console. (_ClipDebug() only writes to the Clipboard.)

When ^ is in $location, the Clipboard is updated each time any of the UDFs is called.

Names This parameter is a string, a comma-separated list of names, optionally preceded by a special character, | . This character changes the delimiters of reported strings, from single quotes to vertical broken bars (¦), making it easier to see leading and trailing spaces.

Value arguments They can be any AutoIt expression that returns a value. There must be a value for each element of the comma-separated list in `$names`. So there are pairs, *name-value pairs*.

Square brackets in $names A *name* can be almost anything, but when the value argument is an array or a map, square brackets in the corresponding *name* can be used to tell cDebug what to report, i.e. to filter out irrelevant elements. For example: if the value argument is a 1-d array, the corresponding name can be `fuddle` or `hunt[2]` or `foo`, but not `vec[` or []. If the value argument is s scalar (simple variable), square brackets can be used freely – in pairs. Uses of square brackets where the value argument is an array or map are described below.

Two simple examples

This script:

```
#AutoIt3Wrapper_Version=B
#include "CDebug.au3"
Local $a=23,$b=2.4
_GuiDebug(@scriptLineNumber,'$a,$b',$a,$b)
```

writes this to the GUI with the title `3`:

```
$a => <Int32>        23

$b => <Double>       2.4
```

and

```
#AutoIt3Wrapper_Version=B
#include "CDebug.au3"
Local $vec=[23,'goat']
Local $ar = [['cat',13],['dog',89]]
_GuiDebug('animal test','obstinate,pets',$vec,$ar)
```

writes to the GUI with the title `animal test`

```
obstinate
obstinate Vector: 2 elements
      [0] => <Int32>     23   0x17
      [1] => <String> (4 chars)   'goat'

pets
pets Array: 2 rows 2 columns
      [0][0] =><String> (3 chars)   'cat'
      [0][1] =><Int32>     13   0x0D
      [1][0] =><String> (3 chars)   'dog'
      [1][1] =><Int32>     89   0x59
```

This demonstrates reporting the values of vectors and arrays. (Here I use *vector* when an array as one dimension.) I would not normally code a name as other than the name of a variable, but it can be done.

From this point on, I will omit the title of the GUI from the results. In all cases, it is the first parameter. When `_ConsDebug()` or `_ClipDebug()` is called, the first parameter appears as the first line, as it does when with `_GuiDebug` **Copy data** is clicked.

Arrays

cDebug reports on 1-d, 2-d and 3-d arrays. The same features are available for all three. To cDebug, 3-dimensional arrays have sheets, rows and columns. No examples are shown for 3-d arrays; they are handled in the same way as 1-d and 2-d arrays. 1-d arrays are called *vectors*.

Array elements that are arrays The UDFs understands elements of arrays that are themselves arrays, so

```
#AutoIt3Wrapper_Version=B
#include "CDebug.au3"
Local $subVec = ['cat','dog']
Local $vec = [$subVec]
_GuiDebug('animals','$vec',$vec)
```

writes to the GUI:

```
*$vec
$vec Vector: 1 element
      [0] =>  Vector: 2 elements
            [0] => <String> (3 chars)   'cat'
            [1] => <String> (3 chars)   'dog'
```

And now for a slightly more complex example:

```
#AutoIt3Wrapper_Version=B
#include "CDebug.au3"
Local $tStruct = DllStructCreate('int;char[7]')
DllStructSetData($tStruct,1,2018)
DllStructSetData($tstruct,2,'Hello')
Local $vec = [$tStruct,DllStructGetPtr($tStruct)]
_GuiDebug(@ScriptLineNumber,'$vec',$vec)
```

produces:

```
$vec
$vec Vector: 2 elements
      [0] => <Struct> (12 bytes) @:0250C3A0  -*- DETECTED -*-
      1  s4byte (int)          => <Int32>     2018   0x07E2
      2  char[7]               => <String> (5 chars)   'Hello'
      <padding>  1 byte
      [1] => <Ptr>       0x0250C3A0
```

with a note at the bottom telling you that it can't differentiate between an `int` and a `long` or a `bool`. See *DLL Structures* below for more on structures; also see below on `<padding>`.

Slices of arrays The main UDFs are also able to dump portions of vectors and rectangular portions of arrays; I call them *slices*.

To dump a slice:

- Code a vector (or array) as a value parameter
- In the corresponding list element of `$names`, after the array name, write `[` ,then an *index expression*, then `]`
- If the array has 2 dimensions, add `[` ,then an index expression, then `]`

Given `$ar[34]`, `34` is an example of the simplest form of a cDebug index expression.

An index expression is made up of one or more index sub-expressions separated by commas, e.g. [23,35] . This is an extension of AutoIt's syntax.

There are five kinds of index sub-expressions:

- Range from, e.g. `23..`
- Range to, e.g. `..45`
- Range from and to, e.g. `23..45`
- Several values, e.g. `23,45`
- a single value, e.g. `23`

Sub-expressions can be combined, for example `21..24,33` . `_cDebug_Example()` demonstrates various possible index expressions.

As well as literal constants, members of index sub-expressions can be variables or declared as constants, e.g. `$a..45` .

For a variable or constant to work in this context, it must be global, be local and declared outside any function, or be an element of `$names` (with a corresponding value parameter).

For 2D arrays, there is also a special sub-expression: `..` (2 periods) : meaning *all indices*. For example, with `$ar` as the value parameter and `$ar[..][2]` as the $name parameter, `$ar[0][2]`, `$ar[1][2]`, etc., i.e. the whole third column, is dumped. Empty square brackets are equivalent to two periods .

A more complex example:

```
#AutoIt3Wrapper_Version=B
#include "CDebug.au3"
Global $g=4,$h=2
foo()

Func foo()
     Local $ar =[[0,1,2,3],[10,11,12,13], [20,21,22,23], [30,31,32,33], [40,41,42,43], [50,51,52,53]]
     Local $kk=3,$m=6.73
     _GuiDebug('test','$kk,$ar[$kk..,$g][$h..],$m',$kk,$ar,$m)
EndFunc
```

writes to the GUI:

```
***test-------------------------
$kk => <Int32>     3   0x03

$ar[$kk..,$g][$h..]
$ar Array: 6 rows 4 columns total
      [3][2]    => <Int32>     32   0x20
      [3][3]    => <Int32>     33   0x21
      [4][2]    => <Int32>     42   0x2A
      [4][3]    => <Int32>     43   0x2B
      [5][2]    => <Int32>     52   0x34
      [5][3]    => <Int32>     53   0x35
      [4][2]    => <Int32>     42   0x2A
      [4][3]    => <Int32>     43   0x2B

$m => <Double>    6.73
```

Note that `$g` does not need to be a parameter but `$kk` does.

Maps

cDebug can report all elements of maps, with their keys. It also permits filtering of map elements by applying a regular expression to the keys, and filtering using the same extensions to AutoIt as for array slices.

If a value argument is a map, three cases are supported: :

- The corresponding name does not have square brackets
- Filtering using a regular expression
- Filtering using the same notation as it used for array slices.

In the first case, all elements of the map are reported.

Filtering using a regular expression After the `[`, code `regexp:` then the regular expression.

For example,

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $map[]
$map.ax = 'aa'
$map.ay = 'bb'
$map['*'] = 'many'
_GuiDebug('test','map[regexp:a]',$map)
```

writes to the GUI:

```
map[regexp:a]
map Map: 3 elements total
      ['ax'] => <String> (2 chars)   'aa'
      ['ay'] => <String> (2 chars)   'bb'
'
```

regexp: tells cDebug to expect a regular expression, not a map key.

Multi-level filtering

Multi-level filtering provides a means of having cDebug report only the elements of arrays and maps in which you are interested.

In the $name argument, -> is used to indicate the hierarchy. The value argument forms a name-value pair with the name of the array or map at the top of the tree. Filtering can be used at one level, and not at another. For example,

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $vecss = [123,'subsub',23,2,-2]
Local $vecs = [567,'cat',$vecss,'sub']
Local $vec = [123,$vecs,'cat'&@CRLF&'dog',789]
_guidebug('test','$vec[1]->vecs[2]->$vecss',$vec)
```

produces:

```
$vec[1]->vecs[2]->$vecss
$vec <Array> Ubound 4
      [1] => vecs <Array> Ubound 4
            [2] => $vecss <Array> Ubound 5
                  [0] => <Int32>     123   0x7B
                  [1] => <String> (6 chars)   'subsub'
                  [2] => <Int32>     23   0x17
                  [3] => <Int32>     2   0x02
                  [4] => <Int32>     -2   0xFFFFFFFE
```

Here $vec is the value argument for $vec[1]->vecs[2]->$vecss .

The names of the arrays and maps are only used by cDebug for reporting, so the $names argument in this example can be written as '$vec[1]->[2] .

While the above example shows only filtering of vectors, maps can also be filtered, as can mixed trees of arrays and maps.

The following example reports the name and colour of each cat:

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $cats[],$cat1[],$cat2[]
$cat1.name = 'jack'
$cat1.colour = 'black'
$cat1.chases = 'squirrels'
$cat1[50] = 5
$cat1[7] = 20
$cats.cat1 = $cat1
$cat2.name = 'suki'
$cat2.colour = 'grey'
$cat2.chases = 'birds'
$cat2.age = 4
$cats.cat2 = $cat2
$cats.cat2.friendly = False
$cats.cat1.friendly = True
_GuiDebug('Test','$cats->["colour","name"]',$cats)
```

The result is:

```
$vec[1]->vecs[2]->$vecss
$vec Vector: 4 elements total
      [1] => vecs Vector: 4 elements total
            [2] => $vecss Vector: 5 elements
                  [0] => <Int32>     123   0x7B
                  [1] => <String> (6 chars)   'subsub'
                  [2] => <Int32>     23   0x17
                  [3] => <Int32>     2   0x02
                  [4] => <Int32>     -2   0xFFFFFFFE
```

You can obtain the name and colour of each of the cats by coding `_GuiDebug('Test','$cats',$cats)` but by using multi-level filtering you can avoid having to search through many lines of report. This is really helpful when arrays and maps have many elements.

There is almost no limit to the number of levels. Multi-level filtering does use recursion, and AutoIt is limited to 63 levels of recursion; cDebug may be limited to fewer levels.

When multi-level filtering is in use, cDebug is much less rigorous in checking for user errors. It reports what is reasonable, If you don't get the result you expect, try backing off to not using this filtering. You will then see why cDebug reported – `no name found with element limit of ...` – . It is likely that the filtering you thought you were asking for is not what you were requesting!

The same sub-expression for an array and a map In AutoIt, a key can look like an array index, so what does cDebug do about it? Consider an example:

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $vec = [100,200,300]
Local $map[]
$map[1] = 'index1'
Local $mainVec = [$vec,$map]
_GuiDebug('test','$mainVec->[1]',$mainVec)
```

produces:

```
$mainVec->[1]
$mainVec Vector: 2 elements
      [0] =>  Vector: 3 elements total
            [1] => <Int32>     200   0xC8
      [1] =>  Map: 1 element total
            [1] => <String> (6 chars)   'index1'
```

It reports both the array element and the map element.

Sub-expression with negative numbers But map keys can be negative integers. What happens when the specified index is negative? Consider this example:

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $vec = [100,200,300]
Local $map[]
$map[-1] = 'index-1'
$map[1] = 'index1'
Local $mainVec = [$vec,$map]
_GuiDebug('test','$mainVec->[-1]',$mainVec)
```

produces:

```
$mainVec->[-1]
$mainVec Vector: 2 elements
      [1] =>  Map: 2 elements total
            [-1] => <String> (7 chars)   'index-1'
```

cDebug correctly reports only map element, ignoring the array.

If the filtering criterion (sub-expression) is changed to `-1..1` cDebug produces:

```
$mainVec->[-1..1]
$mainVec Vector: 2 elements
      [0] =>  Vector: 3 elements total
            [0] => <Int32>     100   0x64
            [1] => <Int32>     200   0xC8
      [1] =>  Map: 2 elements total
            [-1] => <String> (7 chars)   'index-1'
             [1] => <String> (6 chars)   'index1'
```

To accommodate the array, its sub-expression has been changed to `0..1`

DLL structures

cDebug can display the contents of structures, including when the structure is the value of an array element.

How cDebug displays the contents of structures depends on whether you specified a tag in an element of the $names argument, and whether an exclamation mark is coded. cDebug can display either per your tag specification or what it detects from the structure.

You may perhaps wish to skip down this document to the pseudo-code, then return here to read the full explanation.

Coding a tag To include a tag in a name-value pair, append { , the tag, and } to the name element.

An example with a tag as part of the `name element`:

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $tag = 'byte able;float[2];double charlie;bool dog'
Local $tStruct = DllStructCreate($tag)
Local $pi = 3.141592653589793
DllStructSetData($tstruct,'able',23)
DllStructSetData($tstruct,2,$pi,1)
DllStructSetData($tstruct,2,180/$pi,2)
DllStructSetData($tstruct,'charlie',ATan(-1)*360/$pi)
DllStructSetData($tstruct,'dog',True)
_GuiDebug('A test','tstruct{'&$tag&'}',$tStruct)
```

After confirming that the tag coded in the call to `_GuiDebug()` is what cDebug detected, this script produces:

```
***A test-------------------------
tstruct => <Struct> (32 bytes) @:026644A0  -*- PER CALL -*-
       1  byte          => <Int32>     23   0x17
       <padding>  3 bytes
       2  float[2]
                      [1] => <Double>    3.14159274101257
                      [2] => <Double>    57.2957801818848
       <padding>  4 bytes
       3  double        => <Double>    -90
       4  bool          => <Int32>     1   0x01
       <padding>  4 bytes
```

In this example, the name element to which `{'&$tag&'}'` is appended to the name of the structure variable, but the name itself can be anything. For example,

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $tag = 'char a[3]'
Local $tStruct = DllStructCreate($tag)
DllStructSetData($tStruct,1,'cat')
Local $vec = [$tStruct]
_GuiDebug('A test','vec{'&$tag&'}',$vec)
```

produces:

```
vec
vec Vector: 1 element
      [0] => <Struct> (3 bytes) @:0271A998  -*- PER CALL -*-
      1  char[3]   => <String> (3 chars)   'cat'
```

Without coding a tag Now the above example but without a tag:

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $tag = 'byte able;float[2];double charlie;bool dog'
Local $tStruct = DllStructCreate($tag)
Local $pi = 3.141592653589793
DllStructSetData($tstruct,'able',23)
DllStructSetData($tstruct,2,$pi,1)
DllStructSetData($tstruct,2,180/$pi,2)
DllStructSetData($tstruct,'charlie',ATan(-1)*360/$pi)
DllStructSetData($tstruct,'dog',True)
_GuiDebug('A test','struct',$tStruct)
```

produces:

```
struct => <Struct> (32 bytes) @:02633510  -*- DETECTED -*-
       1  u1byte (byte)          => <Int32>     23   0x17
       <padding>  3 bytes
       2  float[2]
                              [1] => <Double>    3.14159274101257
                              [2] => <Double>    57.2957801818848
       <padding>  4 bytes
       3  double                 => <Double>    -90
       4  s4byte (int)           => <Int32>     1   0x01
       <padding>  4 bytes

Note
----
Structs are marked DETECTED if:
-   No tag is specified or
-   A tag is specified and ! follows {, forcing detection.
When such structs are detected:
-  u1byte (unsigned 1-byte) is often byte but it may be boolean
-  s4byte (signed 4-byte) is often int but may be long, bool, int_ptr, long_ptr, lresult or lparam
 lparam
```

The structure not being provided to cDebug, it has **detected** it.

The results are the same except for the type of the last element: it should be a `bool`. cDebug has detected the elements in the structure from the values returned by `DllStructGetData()`; this function returns an Int32 for both `int` and `bool`. Both are signed 4-byte, reported as `s4byte`. In parentheses cDebug shows the most common use of a signed 4-byte.

Also note that the names of the elements are missing: they are not present in the call to `_GuiDebug()`.

Specifying a tag has the following advantages:

- Names of elements are reported, where they are specified in the tag
- The element types will always reflect the specification of the tag (unless `!` is used: see below)
- `struct`, `endstruct` and `align` are reported

So why not always specify a tag? The first answer: speed, i.e. <u>your</u> speed!. If you coded `$tstruct = DllStructCreate(' ...')` , fishing out the tag to have cDebug detect, takes time, your time!.

But there is another reason: what if you are not sure what the tag should be? Perhaps you think either MSDN or the guy whose code you borrowed is wrong. Or perhaps you aren't that good at converting Microsoft's structures to AutoIt tags. Calling one of the main UDFs without a tag allows you to check, but there is an easier way: see below.

Limitations of detection Not specifying a tag means that cDebug can determine whether a 2- or 4-byte numeric element is signed or not, and its length, but there is no way in AutoIt to determine whether an 8-byte integer is signed or not. cDebug displays the most likely type. For other types of elements, what cDebug's detection algorithm reports may depend on whether AutoIt is 32-bit or 64-bit:

Type detected	But it can mean:		
	32-bit and 64-bit	**32-bit**	**64-bit**
`int64`	`uint64`		`int_ptr, long_ptr, lresult, lparam, uint_ptr, ulong_ptr, dword_ptr, wparam`
`int`	`long, bool`	`int_ptr, long_ptr, lresult, lparam`	
`uint`	`ulong, dword`	`uint_ptr, ulong_ptr, dword_ptr, wparam`	
`ushort`	`word`		
`ptr`	`hwnd, handle`		

`ubyte` is a special case: it is not a type, even at Microsoft, because a `byte` is always unsigned. If you, or someone else, codes a `ubyte` element in a structure, cDebug considers it to be a `byte`. Unfortunately, `ubyte` has been in AutoIt's help for `DllStructCreate()` for some years.

Whether or not a tag is specified, to the right of the `=>` the result of `DllStructGetData()` is displayed; however there is one element type for which this function can return different AutoIt *types*[1] depending on the presence (or absence) of its *index* parameter. This element is `byte`:

- `DllStructGetData($tStruct,`*`element`*`)` returns the value of the whole element as a Binary
- `DllStructGetData($tStruct,`*`element`*`,`*`index`*`)` returns an Int32

cDebug does not report both for an element; rather it displays a `binary` if the element has more than the element limit, and `int32` values if there are less.

What if the detected structure does not match what you specified? CDebug tells you before reporting on the structure you specified. If an element of the tag you specified extends beyond the structure you coded, cDebug tells you about the error, then aborts yur script.

When you specify an incorrect tag With a tag present, cDebug always checks whether the structure detected fits the specified tag. If the structure found by cDebug and the one you specified differ, cDebug reports the difference. If the tag you specified creates a structure longer than the one you specified, cDebug tells you, with the note `**?* Element extends beyond structure *?*`

This is an example of a tagged call to `_GuiDebug()` that illustrates this:

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $tag = 'char a[3]'
Local $tStruct = DllStructCreate($tag)
_GuiDebug('test','struct{char [4]}',$tStruct)
```

When this script is run, cDebug warns about the 4 versus the 3, then warns that the element extends beyond the end of the structure, then produces:

```
struct => <Struct> (3 bytes) @:02D11A48  -*- PER CALL -*-
      1  char[4] => <String> (0 chars)   ''
**?* Element extends beyond structure *?*
```

Overriding a specified tag It is possible to override presence of a tag in a call, causing cDebug to report the detected structure. You could see the detected structure by removing `{`, the tag and `}`, but there is a faster way: just insert a `!` (exclamation mark) after the `{` .

The above example would then be:

---

[1] AutoIt has only one type, *variant*. This type has *subtypes*, including `binary` and `int32`.

```
#AutoIt3Wrapper_Version=B
#include "cDebug.au3"
Local $tag = 'char a[3]'
Local $tStruct = DllStructCreate($tag)
_GuiDebug('test','struct{!char [4]}',$tStruct)
```

and the results in:

```
struct => <Struct> (3 bytes) @:0278D5E0  -*- DETECTED -*-
      1   char   => <String> (0 chars)   ''
```

Padding AutoIt has no any way of detecting `DLLStructCreate`'s `struct`, `endstruct` and `align` (because AutoIt can't detect them), but their effects can be seen in the existence of padding bytes. cDebug does report `struct`, `endstruct` and `align`, when they appear in a tag specified in a call to any of the main UDFs. It always detects padding bytes.

Pseudo-code of per-call and detected reporting

```
Detect the structure, store a report on it, also return an array containing element and padding data
If there is a tag in the cDebug call then
     If this tag begins with ! then
          Remove !
          Show the detected report
     Else
          Generate a per-call report, also return a similar data array
          Compare the detected and per-call data
          If they differ then
               Tell the user that they differ
          EndIf
          Show the per-call report
     EndIf
Else
     Show the detected report
Endif
If detected report shown for any of the name-value pairs, append to it limitations of detection
```

Easy coding

The format for calling the UDFs has been designed to make coding a call as fast as possible, minimizing coding effort and the chances of errors: the $name argument is often the same as the variables arguments, enclosed in quote marks. Here is how I do it:

- Type `_GuiDebug(@Sc` then use Auto-complete to choose `@ScriptLineNumber`
- Type , '',
- Use Auto-complete to help in typing variable names (and Copy/Paste for expressions)
- Type the closing `)`
- Back-arrow once
- Control_Shift_BackArrow back to after the comma
- Key Ctrl_C, move the cursor to the `$name` parameter
- Key and Ctrl_V.
- If you are using slices, type the square brackets and the expression.

This may look complicated, but I seem to have developed local intelligence in my fingers: they seem to know mostly what to do!

Changing a call from one of the functions to another requires minimal effort. For example, if you have been dumping to a GUI and decide to change to dumping to the Console, the only change needed is to type `Cons` in place of `Gui` . Switching to `_ConsDebug()` is handy for tracing trouble in a loop.

Copying location and data to the Clipboard

There are two ways of copying results to the Clipboard:
1. by clicking on `Copy data` in the GUI, and
2. by inserting ^ into `$location`

Both ask whether the Clipboard should be cleared before copying. When using `_GuiDebug()`, I use the convenience of the `Copy data` button.

Conditional debugging

Write `Global $gbDebugging = True` to trigger debugging, and `If $gbDebugging Then GuiDebug (...)` where variables are to be dumped. (There is nothing special about the name `$gbDebugging`.)

Changing the number of elements displayed temporarily

There are two ways of changing the number of elements of arrays and maps, and number of dimensioned structure elements that are displayed:

- when running `_GuiDebug()`, by clicking on the `Change element limit` button, and
- by coding `_ChangeElementLimitForDebug(...)` before calling one of the debug UDFs.

When running `_GuiDebug()`, if you choose a different limit, the results are immediately redisplayed with the new limit. `_ChangeElementLimitForDebug()` takes one argument: the new limit.

Out of the box, values of up to 30 elements of each dimension of an array, up to 30 values of each dimensioned structure element, and up to 30 elements of a map, are shown. The limit also affects how much of a binary value or string is shown. If there are more elements than this, `...` *n* `more elements` is displayed, or with filtering, ... *n* `more selected elements`.

An example:

- You ask `_GuiDebug()` to write a vector (a 1-d array) `$vec` with 28 elements to the GUI. Out of the box, 30 element values will be displayed.
- This is a long list, so you decide only display 10 values.
- Before `_GuiDebug('An array','$ar',$ar)`, you click on `Change element limit` and enter `10`.. Now `_GuiDebug` displays the first 10 elements. It then displays `... 18 more elements`. (If you are outputting to the Console, `_ChangeElementLimit(10)` does the same thing.)

If you then realize that the elements you are interested in begin at the fifth element, you code `_GuiDebug('A vector','$vec[4..]',$vec)`

Because the limit is still 10, `_GuiDebug()` reports the values of the elements 5 to 14 (ref 0).

Limits work with array slices and maps too.

Customizing cDebug permanently

There are several ways in which you can customize how cDebug behaves, by changing the values of global variables in cDebug.au3 but whenever you download an update to cDebug, you need to make your changes again.

There is a better way:

- Create a .au3 file containing your changes. For example, create `mycDebug.au3` containing `Global $g_cDebug_bAlwaysAskClearClipboard = True`
- Place `#include "mycDebug.au3"` before `#include"'cDebug.au3"`

You can also just place `Global $g_cDebug_bAlwaysAskClearClipboard = True` in your script, before `#include"'cDebug.au3"` .

By default, cDebug reports comparison of your specified structure tag with the detected tag only if they differ significantly.. If you wish the comparison to be always displayed, change `$g_cDebug_ShowStructComparisonAlways` from `False` to `True`. **This is a possibly script-breaking change.** This flag replaces `$g_cDebug_bTellStructComparisonIsReasonable`.

How to change how many elements of arrays, maps and structure elements are shown is described in the previous section. To change this setting permanently, set $g_cDebug_elementLimitDefault to a different positive integer or, to show all elements, set it to 0

By default, the next time you run a script that calls one of the main cDebug UDFs, the element limit will be the limit you last used in the same *scope*. If you wish to have the limit revert to $g_cDebug_elementLimitDefault, set $g_cDebug_bRetainElementLimit to False.

What does *scope* mean in this context? When you script finishes (or aborts), the element limit then in effect is saved to cDebug.ini in a folder. The default *scope* is the folder in which your script resides: $g_cDebug_iniFolder = ''. If you set $g_cDebug_iniFolder to a specific hard-coded folder, the limits will be same for all your scripts, independent of what folder they are in.

If you leave $g_cDebug_iniFolder set to '', and call a main UDF from scripts in various folders, there will be an element limit (and cDebug.ini) for each folder.

Most of the horizontal spacing in reports depends on the setting of $g_cDebug_Indent. Out of the box, this variable has a value of 6. You can expand or contract lines by changing the value to another (small) positive integer.

Out of the box, when @CRLF, @CR and @LF occur in a string, the line ends are reported and the string shows as multiple lines. This is because $g_cDebug_bDoLineBreaksInStrings defaults to True. If it is set to False, @CRLF, @CR and @LF are reported, as are other control characters. In the default case, the number of lines reported is governed by the element limit; in the other case, how many characters of a string are shown is governed by the element limit.

When you visit the Change Element Limit dialog, the second choice is set the element limit really large. The number of elements offered there initially is 1000. You can change this number by changing the value to which $g_CDebug_LargeElementLimit is set.

By default, when you click on Copy to Clipboard, if text is on the Clipboard, when cDebug reports for the first time in the run of your script, cDebug asks whether to clear the Clipboard before copying your data to it. (It does not ask if the Clipboard contains non-text.). To have cDebug ask you every tine cDebug reports, set $g_cDebug_bAlwaysAskClearClipboard to True.

The default is for strings to be surrounded by single quote marks. This can be changed by setting $g_cDebug_stringDelim to a different character.

When you are finished debugging

Because the names of the UDFs all contain Debug, just search for occurrences of Debug in your script. Most (if not all) of these lines will be calls to one of the UDFs. Then remember to remove the #include cDebug.au3 line at the top of your script.

Error detection

cDebug detects user errors. When it finds an error, it reports it in a dialog, including $location, $names and, where pertinent, the particular name element. It then aborts your script. If you find a user error it doesn't catch, tell me, c.haslam, on the forum!

It also checks for some possible coding errors, and invites you to report them. At this time, no coding errors are known, but the possibility exists in any software, especially when the tester is the same person as the programmer!

Naming of functions and global constants and variables

All internal functions in cDebug begin with `_cDebug_`. All global variables begin with `$g_cDebug_`.

Because calling the example function `_Example` may lead to name conflicts, it is called `_cDebug_Example`. There is a call to it at the end of `cDebug.au3`, which is commented out.

Files

- cDebug.au3
- cDebug.ini – created and updated if `$g_cDebug_bRetainElementLimit` is `True`. See above for its possible locations.

Caveats

At this time, there is only one caveat, borne of daily use of cDebug.

I had written code like this in a script I was developing:

```
Local $hWin = GUICreate(...)
.
.
.
Initialize()
.
.
.
GuiSetState()
While 1
     Switch GUIGetMsg()
          Case $GUI_EVENT_CLOSE
               .
               .
               .
     EndSwitch
```

In `Initialize()` I called `_GuiDebug()`. The GuiDebug dialog box showed properly. I then clicked on OK to continue running my script. TrayIconDebug showed that the script was at the `EndSwitch` line shown above. The dialog box did not appear.

Running my script without the GuiDebug() call worked AOK.

Changing `GuiSetState()` to `GuiSetState(@SW_SHOW,$hWin)` solved the problem. This behaviour does not seem to be due to a bug in cDebug.

Known limitations

1. If you call any of the main UDFs with a value of `1e308`, that parameter and all value parameters after it, will be ignored. cDebug can't use `Default` as the default value because it needs to count the value parameters and report a value that is `Default` (for example, when a value argument is a non-existent map element). `@NumParams` is not used because it seems to be unreliable when a value parameter is a structure, hence `1e308`

2. The limitations of detecting DLL-structure elements are discussed above.

3. Writing a string to an Edit box that contains `@LF` causes the rest of the string, and the preceding part of the string, back to the last `@CR` or `@CRLF`, not to be written. So while setting `$g_cDebug_bDoLineBreaksInStrings` to `True` (the default) causes `@CRLF` and `@CR` in a string to write `@CRLF` and `@CR` to the Edit box, `@LF` in a string causes `@CRLF` to be written to the box.

4. Consider this code:

```
#include 'cDebug no maps.au3'
Local $aSub = [1,2]
Local $aMain = [10,20,30,40,$aSub]
_ConsDebug(@ScriptLineNumber,'$aMain[4]',$aMain[4])
```

produces:

```
***cDebug Error-------------------------
In $aMain[4], element number exceeds 1, the maximum
Location: 4
Names:    $aMain[4]
Current name: $aMain[4]
Aborting the script
```

This is happening because cDebug is picking up the size of `$aSub`, not the size of `$aMain`: `$aMain[4]` is `$aSub`. cDebug does not have access to `$aMain` in this call of `_consDebug()`. The solution is code `_ConsDebug(@ScriptLineNumber,'$aMain[4]',$aMain)`

5. When a user clicks on a button created by his code, and a _GuiDebug dialog box is underneath it with a button that is in the same screen position as the user's button, the _guiDebug button gets clicked instead. There is no known solution to this bug.

Edit history

1.0.0	First release to AutoIt forum
1.1.0	Added _GuiDebug()
1.2.0	_GetCtrlFontSize() determines font name and point size; internal functions renamed to _cDebug_*
1.3.0	Added subtypes so now handles all Autoit subtypes; improved layout of results; fixed 2 bugs in Slices
1.4.0	Added display only a limited number of elements; added __ChangeElementLimitForDebug and the Change element limit button; detects more user errors in Slices
1.4.1	For binary variables with long values, shortened displayed value
1.4.2	Fixed Copy Data button not disabling after copying to Clipboard
1.4.3	Removed false detection of user error
1.5.0	Added ability to display structs with a tag specified, and without
1.5.1	Commented out call to `_cDebug_Example()`; Incorporated _StringSize() so no more #include "StringSize.au3"
1.5.2	Added reporting `struct`, `endstruct` and `align`; fixed string value containing single quote mark
1.5.3	Removed duplicate `#include-once` at line 1534
1.6.0	Changed the default for value parameters; added info messages when comparing a user tag and a determined tag; improved logic for comparing tags
1.6.1	Various small bug fixes; Note re detection now per main UDF call was per name-value pair; parent GUIs now lose focus properly
1.6.2	Removed ! From `_cDebug_Example()`
1.7.0	Rewrote comparison of specified structure with detected structure
1.7.1	When comparing detected and per-user structures, treats uint64 as int64; fixed @CRLF problem in reporting detected structure
1.7.2	Comparison results: reformatted, added element length
1.8.0	Dialogs avoid hiding code in SciTE as much as possible
1.8.1	Allows using cDebug without SciTE visible (again)
1.8.2	Error and Warning message boxes lacked `@CRLF&@CRLF`
1.9.0	Rewrote structure comparison; added reporting maps
1.9.1	Fixed bug in nesting maps within arrays
1.9.2	Fixed bug in nesting arrays in maps
1.9.3	Maps: cosmetic display bug fixed; key sort sequence now case sensitive; added ability to suppress *structure comparison is reasonable* dialog
1.10.0	Added retention of element limit; added no element limit; added scope of element limits; indenting now customizable; added expansion of dialogs to full screen width

1.11.0 Fixed indentation for certain vector elements; added line breaks in string display; added partial maps
1.11.1 Fixed case where cDebug.au3 included but no main UDF called
1.11.2 Fixed $name is map element
1.11.3 Fixed array[index] in $name
1.11.4 Fixed bug in _cDebug_FormatNameAndValue_VectorSlice()
1.11.5 Fixed missing & in &= in calling _cDebug_FormatNameAndValue_VectorSlice()
1.11.6 Fixed => bug in _cDebug_FormatNameAndValue_VectorSlice()
1.12.0 Added multi-level filtering
1.12.1 Fixed bug in regexp for array slices
1.13.0 Added multi-level filtering for maps
1.13.1 Fixed errant user error for multi-level in e.g. [20..] if vector has less than 20 elements but map has map[20]
1.13.2 Fixed bug re reporting map containing no elements
1.13.3 Added check that $names parameter is a string
1.14.0 Rewrote multi-level filtering; for detected structures, element types are now reported as signed/unsigned-size with most likely types
1.14.1 Fixed bug in _cDebug_FmtNameVal_ArraySlice()
1.14.2 Fixed second bug in same function
1.15.0 Added 3-d arrays; improved element-limit reporting code; added cDebug no maps.au3
1.15.1 Fixed bug in sub-expression
1.15.2 Added $g_cDebug_bAlwaysAskClearClipboard and ability to customize cDebug without changing cDebug.au3; _cDebug_Example() now runs again with maps
1.15.3 Reatored showing element name for user-specified tags; improved parsing of struct elements; changed flag from $g_cDebug_bTellStructComparisonIsReasonable to $g_cDebug_ShowStructComparisonAlways
1.15.4 Independent of GuiOnEvent mode; Added missing parameter to parameter dump; Meets full AuCheck requirements; Added accelerators for buttons; Including more than once works

Acknowledgements

I am indebted to Melba23 for his `_StringSize()`, to Kafu and ProgAndy for `_GetCtrlFontSize()`, to jchd for the code for subtypes that cDebug did not previously handle, and for help via the forum. Also thanks to Jos van der Zande, LazyCoder, Tylo and Ultima, the authors of `__ArrayQuickSort1D()`

c.haslam