



---

# Rocket Uniface Library 10.4

---

# Notices

## Copyright

© 1996-2025 Rocket Software, Inc. or its affiliates. All Rights Reserved.

## Trademarks

Rocket is a registered trademark of Rocket Software, Inc. For a list of Rocket registered trademarks go to: [www.rocketsoftware.com/about/legal](http://www.rocketsoftware.com/about/legal). All other products or services mentioned in this document may be covered by the trademarks, service marks, or product names of their respective owners.

## Examples

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

## License agreement

This software and the associated documentation are proprietary and confidential to Rocket Software, Inc. or its affiliates, are furnished under license, and may be used and copied only in accordance with the terms of such license.

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

---

# Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: [www.rocketsoftware.com](http://www.rocketsoftware.com)

Rocket Global Headquarters

77 4th Avenue, Suite 100

Waltham, MA 02451-1468

USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

## Country and Toll-free telephone number

- United States: 1-855-577-4323
- Australia: 1-800-823-405
- Belgium: 0800-266-65
- Canada: 1-855-577-4323
- China: 400-120-9242
- France: 08-05-08-05-62
- Germany: 0800-180-0882
- Italy: 800-878-295
- Japan: 0800-170-5464
- Netherlands: 0-800-022-2961
- New Zealand: 0800-003210
- South Africa: 0-800-980-818
- United Kingdom: 0800-520-0439

## Contacting Technical Support

The Rocket Community is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Community and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Community or to request a Rocket Community account, go to [www.rocketsoftware.com/support](http://www.rocketsoftware.com/support). In addition to using the Rocket Community to obtain support, you can use one of the telephone numbers that are listed above or send an email to [support@rocketsoftware.com](mailto:support@rocketsoftware.com).

---

# Table of contents

<b>Structs</b> .....	<b>5</b>
Structs and Members .....	7
Struct Leaves .....	9
Struct Annotations .....	12
Struct Variables .....	14
Struct Access Operators .....	17
Struct Dereference Operator (->) .....	18
Struct Index Operator ({N}) .....	20
Struct Collection Operator (->*) .....	21
Struct Functions .....	23

---

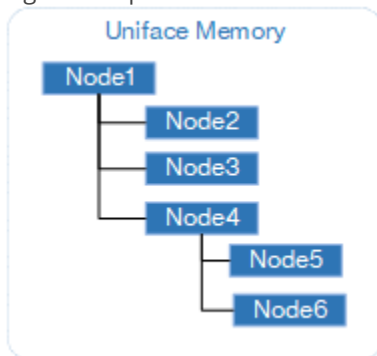
## Structs

A Struct is a tree-like data structure in memory that is used to dynamically manipulate complex data and transform it from or to XML, JSON, or Uniface component data. A Struct can be addressed by assigning it to a variable, parameter, or non-database field of type **struct**, and manipulated using ProcScript commands, access operators, and information functions. For information on using Structs, see [Transforming Complex Data Using Structs](#).

### Structs

A Struct is a collection of data objects, or nodes, that are hierarchically organized under a single root node.

Figure: Simple Struct



A node that has a parent node is a *Struct member*. A member can be a Struct itself, in which case it is called a *nested Struct*.

### Struct Variables

A Struct exists only in memory, and it exists only as long as there is a variable, parameter, or non-database field of type **struct** that refers to it (or one of its members).

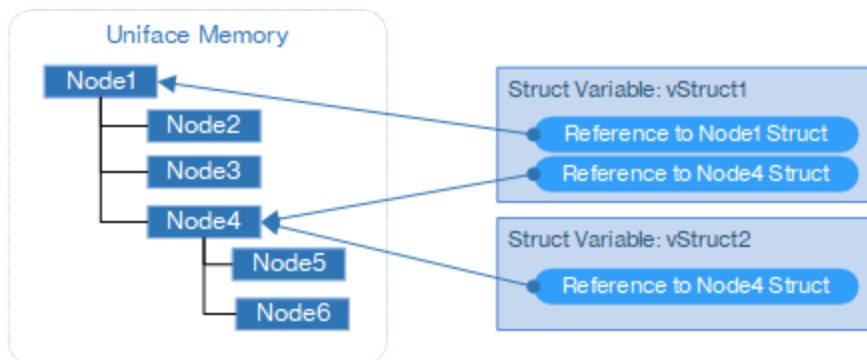


**Note:** For purposes of discussion, the term *struct variable* is usually used, but it can equally apply to **struct** parameters and non-database fields.

The **struct** variable does not contain the value. Instead, it contains zero or more references to Struct nodes.

One **struct** variable can contain multiple references to different Struct nodes, and several variables can refer to one Struct node.

Figure: Struct Variables



For more information, see [Struct Variables](#).

## Creating and Deleting Structs

A Struct can be explicitly created using the `$newstruct` ProcScript function (`vStruct = $newstruct`), or it can be implicitly created in the following ways:

- Assigning a value to a **struct** variable or parameter.
- Assigning a value to an allowed Struct function (`$name`, `$parent`, or `$scalar`) or to `$tags`.

```
vStruct->$name = "My Book"
```

**Note:** Struct access operators, such as the *de-reference* operator (`->`) and the index operator (`{N}`), enable you to address Structs and their members by name and index number. Struct functions enable you to get and set the values and properties of Structs and their members, and insert, copy, move, and remove them. For more information, see [Struct Access Operators](#) and [Struct Functions](#).

- A ProcScript conversion instruction, such as `xmlToStruct` or `componentToStruct` is used. For example:

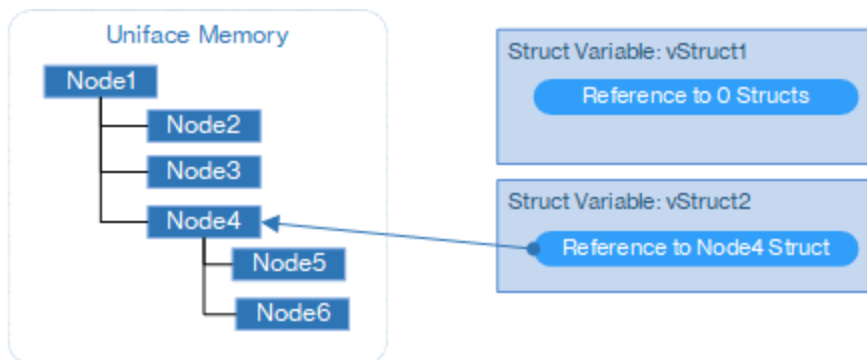
```
xmlToStruct vStruct, "Book.xml"
```

For more information, see [ProcScript for Manipulating Structs](#).

A Struct exists in memory as long as there is a **struct** variable or parameter that points to it. There is no ProcScript command to delete a Struct. Instead, a Struct is deleted as soon as it is no longer referenced by any **struct** variable or parameter, or it is no longer owned by another Struct. For example, reassigning the `vStruct1` variable to another value deletes the Struct it refers to.

```
vStruct1 = ""
```

Figure: Deleted Struct



However, the Struct referenced by vStruct2 continues to exist.

For more information, see [Creating and Deleting Structs](#) and [Assigning Values to Structs](#).

**Note:** Manipulating Structs is memory-intensive. If the Struct is very large or complex, or Struct nodes contain many annotations, Uniface may run out of memory and exit without warning. This depends on the platform and the amount of memory available. For more information, see [Structs and Memory](#).

#### Related concepts

[Working with Structs](#)

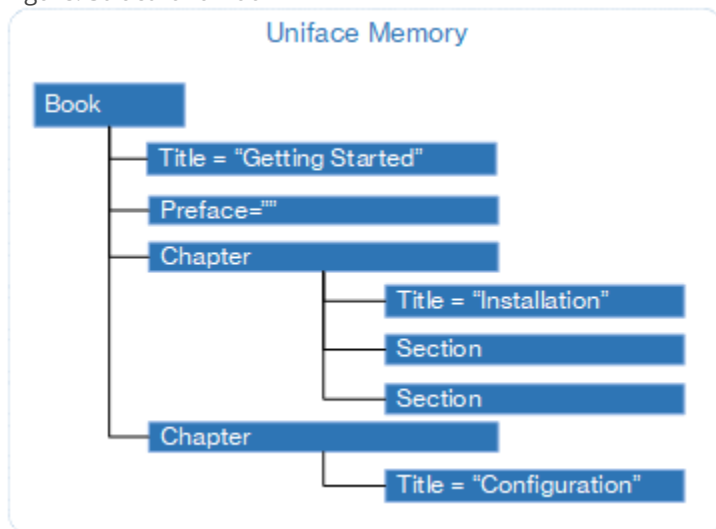
[Struct Annotations](#)

## Structs and Members

A Struct is a data structure in memory, consisting of one top node and zero or more sub-nodes called *members*. Each member can hold a scalar data value (such as a number, string, or date), or it can hold another (nested) Struct.

For example, a book is a complex data structure that can be represented by a Struct.

Figure: Struct for a Book



This illustrates a number of characteristics of Structs:

- A Struct consists of a top-level node and an arbitrary number of sub-nodes, known as *members*.

---

For example, the Book Struct contains members called Title, Preface, and Chapter.

- A Struct member can have a value. This can be:
  - A scalar value, such as numeric, float, string, boolean, date, or time. For example, the value of first Title member is the string "Getting Started".

A Struct member that has no value, or only a single scalar value, is a *Struct leaf*.

- A Struct, in which case it is a *nested Struct*. For example, each Chapter is a nested Struct because it has child members.
- A Struct can hold a mixture of both scalar members and nested Struct members.

For example, the Book Struct contains two scalar members (Title and Preface) and two nested Structs (Chapter).

- A Struct or Struct member can have a name.

If it has a name, the name is case-sensitive. If it is a nested Struct, it is known to its parent by its name. The name can consist of an empty string (""), so an expression such as `vStruct->""` refers to all members that have an empty string as a name.


- Members of a Struct can have the same name.

For example, there are multiple members called Chapter and Section. Thus, if a **struct** variable references a Struct by name (for example `vBook->Chapter`), it contains a collection of references to all Structs with that name. For more information, see [Struct Dereference Operator \(->\)](#).

- Struct members are sequentially ordered.

They can be addressed by their index position, making it possible to address a specific member, even if it has the same name as another member, or no name. For example, `vBook->Chapter{2}` refers to the second chapter member. For more information, see [Struct Index Operator \({N}\)](#).

- A Struct or nested Struct node can have a special `$tags` Struct containing annotations. This Struct is not counted as a normal member of the Struct. In the example, `$tags` is available (but not shown) on the Book Struct, and on the Chapter Structs. For more information, see [Struct Annotations](#).

 **Note:** The term *Struct node* is often used for the top-level Struct and nested Structs, to distinguish them from scalar Struct members. The term refers to the Struct, excluding child nodes but including its properties, such as name and `$tags`.

Most importantly, Structs are dynamic. You can add members to Structs, copy or move members from one Struct to another, and change the values of Struct members. For more information, see [Working with Structs](#).

When working with Structs for XML or XHTML, you may need to add members to Struct leaves, in which case they implicitly become nested Structs with mixed content, or remove members from a nested Structs to make them Struct leaves. For more information, see [Struct Leaves](#).

## Related concepts

[struct](#)

[Transforming Complex Data Using Structs](#)



---

## Struct Leaves

A Struct leaf is a Struct member that has no value, or only a single scalar value. However, adding another value (member) to a Struct leaf implicitly changes it to a nested Struct. The value of the former leaf is now held in a nameless Struct, known as a *scalar Struct*, and the nested Struct has a mixed data structure that may require special handling.

### Value of a Struct Leaf

The value of a Struct leaf is returned when you refer to the Struct member. For example, the following statement puts the value of the member `myLeaf` in the message frame:

```
putmess struct1->myLeaf
```


In fact, the value is treated as a member of the leaf, so the function `$membercount` typically returns `1` for a leaf.

However, nested Structs can also have a single member, and sometimes it is important to distinguish between Struct leaves and nested Structs. In such cases, you can use the Struct function `$isLeaf`, which returns `1` for a leaf, and `0` for a non-leaf.

In most cases, this is all you need to know to understand and work with Struct leaves. However, if the Struct embodies a mixed data content model, which combines scalar values with complex values, you need to be able to distinguish between the scalar and non-scalar members. Mixed content is not possible in Uniface component structures, but it is supported by XML and common in XHTML.

### Scalar Structs

A scalar Struct is used only to contain the scalar value. It cannot have child nodes, and it is not possible to use access operators (such as `->`) on a scalar Struct. To distinguish a scalar Struct from a Struct node, you can use the `$isScalar` Struct function, which returns `1` for scalar Structs and `0` for Struct nodes.

 **Important:** Most ProcScript does not deal with scalar Structs. For example, a process that loops through the children of a Struct, drilling down to the children that are Structs themselves, should stop where it encounters a Struct leaf. During such loops, test for `$isLeaf` and do not request the members of a leaf. Also `$dbgstring` and `$dbgstringplain` do not display the leaf value (a scalar Struct) as an actual Struct, but merely as a value.

When a Struct leaf is assigned an additional Struct member and becomes a nested Struct, the value remains the same. More precisely, the value of a Struct is the concatenated value of all of its direct members that are scalar Structs. Thus, for the following XHTML code, the scalar value would be `Text can be bold or italic`:

```
<div>Text can be <b>bold</b> or <em>italic</em></div>
```

In the following representation, the nameless Struct members are scalar Structs:

```
[ ]
[div]
"Text can be "
[b] = "bold"
"or "
[em] = "italic"
```

You can retrieve all the scalar members of a Struct using the Struct function `$scalar`.

To illustrate the dynamic nature of Struct leaves, consider how to build the following piece of HTML. (part of a table) using Structs.

```
<td>John <b>Smith</b></td>
```

The element `td` has mixed content, containing both text (John ), and an element (`b`) with only text (Smith).

1. First, create a Struct member `td` with value John :

```
mystruct->td = "John "
```

At this point, `td` is a leaf, so `mystruct->td->$isLeaf` is 1

2. Verify this by displaying the Struct in the message frame:

```
putmess mystruct->$dbgstring
```

Because `td` is a leaf, John is displayed as a value:

```
[]  
[td] = "John "
```


3. Now add a member to represent the bold element containing Smith:

```
mystruct->td->*{-1} = $newstruct ; create new member under td  
mystruct->td->*{-1}->$name = "b" ; assign the name  
mystruct->td->b = "Smith" ; assign the value  
putmess mystruct->$dbgstring
```

`td` is now a complex node, so `mystruct->td->$isLeaf` is 0

The message frame shows "John " as a member, rather than as a mere value:

```
[]  
[td]  
"John "  
[b] = "Smith"
```

 **Tip:** A more straight-forward way to build this HTML construct is with the following code:

```
mystruct->td = "John " 1  
mystruct->td->b = "Smith" 2
```

**1** Create a member `td` with value John. At this point, `td` is a leaf, so `mystruct->td->$isLeaf` is 1

**2** Add a member `b` to `td`, with value Smith. Now `td` is a complex node, so `mystruct->td->$isLeaf` is 0

Mixed content Structs can also be created by moving one Struct to another. For example:

```
mystruct->td = "John " 1
tmp->b = "Smith" 2
tmp->b->$parent = mystruct->td 3
```

- 1 Create a member td with value John. At this point, td is a leaf, so mystruct->td->\$isLeaf is 1
- 2 Create another struct with member band value Smith. The Struct tmp is a placeholder on which a new member named b with value Smith is created
- 3 Use \$parent to move the member b to mystruct. By default, the member is appended after the last member, so there is no need to set \$index to 2.

### Effect of XML Attributes on Struct Leaves

Other XML constructs can result in a Struct member being treated as a Struct node instead of a Struct leaf. For example, XML elements can have attributes, which are handled as child elements of their parent element. For example:

```
<div class="note">Text can be bold</div>
```

The \$dbgstringplain representation looks like this:

```
[]
[div]
[class] = note
Text can be bold
```

The following table shows the values returned by Struct functions for the div member, indicating that div is a node with 2 members:

Struct Function	Returned Value
vStruct->div->\$isLeaf	0
vStruct->div->\$isScalar	0
vStruct->div->\$memberCount	2

The following table shows the values returned by Struct functions for the nameless scalar member. It shows that it is a scalar Struct with no members:

Struct Function	Returned Value
vStruct->div->{*2}->\$isLeaf	1

Struct Function	Returned Value
<code>vStruct-&gt;div-&gt;{*2}-&gt;\$isScalar</code>	1
<code>vStruct-&gt;div-&gt;{*2}-&gt;\$memberCount</code>	0

### Related concepts

[\\$isLeaf](#)

[\\$isScalar](#)

[\\$scalar](#)

[\\$memberCount](#)

## Struct Annotations

Struct annotations are descriptive data elements (also known as *tags*) that are used to correctly interpret data in Structs and convert data to and from Uniface component data, XML, and JSON.

Each Struct node has a special child Struct to hold annotations. This Struct can be accessed using the **\$tags** Struct function and is therefore known as the **\$tags** Struct.

Unlike normal Struct members, annotations have no specific position inside the Struct, and are therefore not counted or treated as members of the Struct. This is in contrast to, for example, XML processing instructions, which are treated as normal Struct members because their position in an XML document is relevant, even if they are not part of the document contents.

### Example: \$tags Struct for XML

Consider the following XHTML code:

```
<div class="note">Text can be <b>bold</b></div>
```

When converted to a Struct, each element and attribute is converted into a Struct member, and the `xmlClass` annotation is set to indicate the original XML constructs. This can be clearly seen in the string returned by **\$dbgString** Struct function, in which the **\$tags** Struct (where present) is always the first child of the Struct member:

```
[ ]
[div]
[$tags]
[xmlClass] = element
[class] = note
[$tags]
[xmlClass] = attribute
[ ] = Text can be
[b] = bold
[$tags]
[xmlClass] = element
```

To get the value of a particular annotation you can use the **\$tags** Struct function. For example, the following code assigns the value of the `xmlClass` annotation of the Struct called `div` to a variable:


```
vClass = vStruct->div->$tags->xmlClass
```

For more information, see [\\$tags](#) and [Struct Access Operators](#).

## Annotations for Data Conversion

The ProcScript commands for converting data to and from Structs support fixed sets of tags that are specific to the data format. The annotations typically define the data class or object type and additional metadata unique to the data format.

Because annotations always apply to a specific format, they are ignored when converting to other formats and will be lost. For example, XML annotations are ignored when converting from a Struct to a component.

 **Note:** If annotations are supposed to affect the conversion or the data, you need to ensure that the ProcScript manipulates the Structs based upon the annotations.

When you are creating and populating a Struct in preparation for transformation to another format, you can set your own annotations.

- For **componentToStruct** and **structToComponent**, annotations are used to define the type of Uniface object and the reconnect status of occurrences. For more information, see [Struct Annotations for Uniface Component Data](#).
- For **xmlToStruct** and **structToXml**, annotations are used to indicate the type of XML construct or instruction. For example, the **xmlClass** tag contains information about how the data was stored in the source XML (for example, attribute or element). This is useful when doing a round-trip conversion, so no information is lost. For more information, see [Struct Annotations for XML](#).
- For **jsonToStruct** and **structToJson**, annotations indicate the JSON class and data types. For more information, see [Struct Annotations for JSON](#).

Some data formats have their own metadata, which can be added as annotations. For example, the XML DOCTYPE specification and XML Declaration are required in XML data.

## Annotations for Data Interpretation

Some annotations may help to interpret data correctly. For example, when converting from XML to Struct, the XML Schema may specify that the value was originally of type `Duration`, a type that has no equivalent in Uniface. In this case, the value must be passed as a String but the original data type is added as an annotation so that you can apply logic to handle this data type.

Annotations may also be provided as read-only information to the developer. Such annotations can be ignored on the conversion back. For example, when using an XML Schema, the **xmlDataType** tag is added to specify the primitive type of a scalar element. When converted back using the same XML Schema, **xmlDataType** is ignored. (It is not possible to change the XML Data Type by setting this tag as the data type is determined by the XML Schema.)

## User-Defined Annotations

You can also add your own annotations if you are creating customized conversion routines. In this case, you should ensure that you adopt a naming policy that prevents tag name conflicts with future versions of Uniface. For example, you could use tag names that begin with an underscore ( `_` ).

**Important:** Uniface guarantees that future tag names will never use the underscore as first character.

#### Related concepts

[Struct Annotations for XML](#)

[Struct Annotations for Uniface Component Data](#)

[Structs for JSON Data](#)

[\\$tags](#)

[\\$istags](#)

[\\$dbgString](#)

#### Related tasks

[Example: Tags Inheritance](#)

## Struct Variables

A Struct variable is a variable, parameter, or non-database field of type **struct** that holds an ordered collection of references to zero or more Structs or Struct members.

Struct variables are similar to handles in that both **struct** and **handle** are reference data types that cannot hold a value. Instead they hold references to data in memory. This is in contrast to scalar variables, which hold a value.

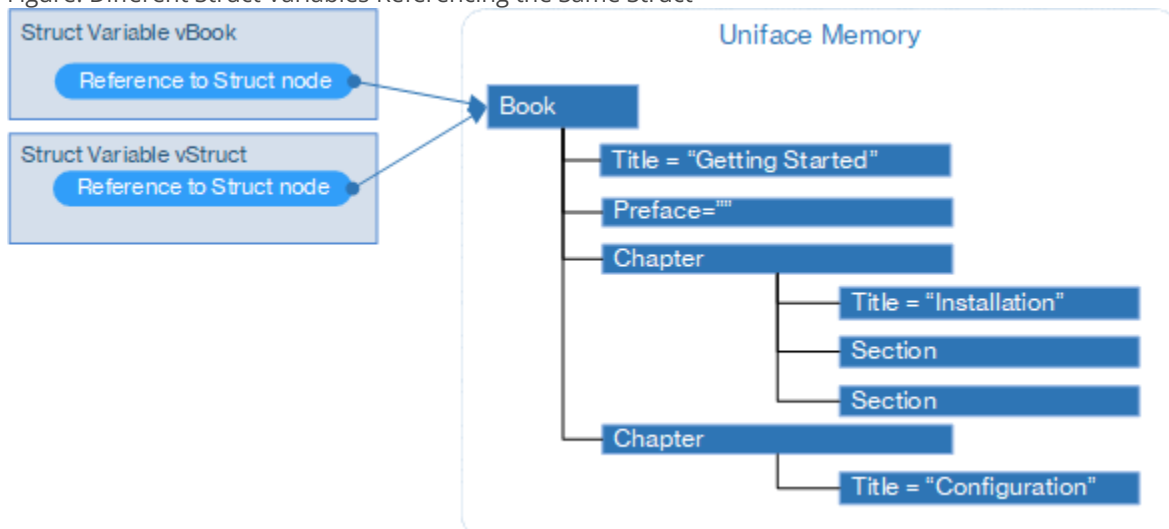
Unlike a **handle**, which can only hold one reference, a **struct** variable holds a collection of references.

The nature of the **struct** data type has a number of consequences when accessing a Struct:

- Several **struct** variables can refer to the same Struct. For example:

```
variables
  struct vStruct, vBook, vChapter, vTitle
endvariables
; Convert an XML document to a Struct
xmlToStruct vBook, "file://book.xml"
vStruct = vBook
```

Figure: Different Struct Variables Referencing the Same Struct

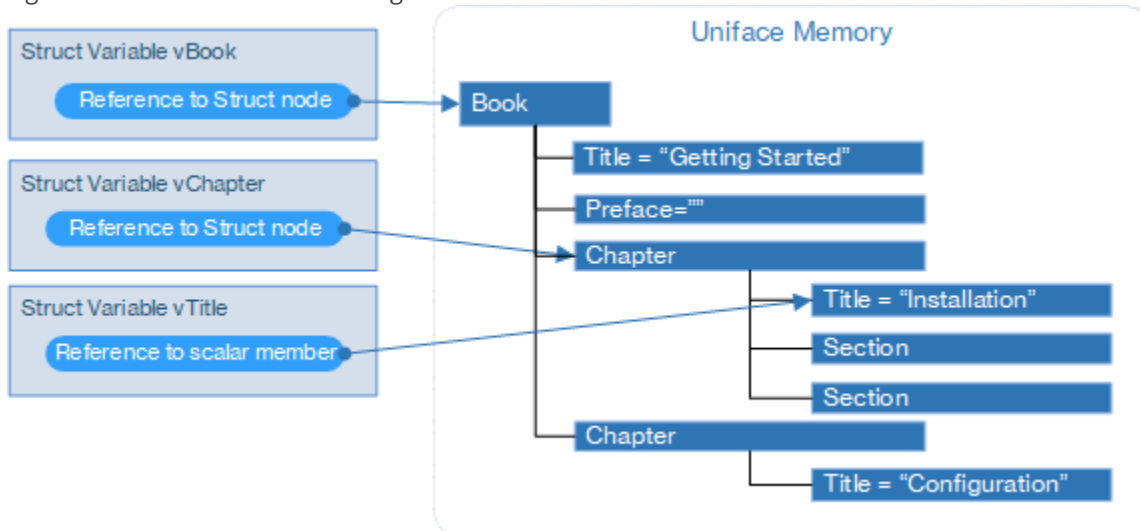


- A **struct** variable can refer to a member of a Struct. Access operators enable you specify the member.

```
vChapter = vBook->chapter{1} ; Struct index operator
vTitle = vChapter->Title ; Struct de-reference operator
```

vChapter now refers to the first member named chapter in the Struct referenced by vBook, and vTitle refers to the title of that chapter. The chapter member is a nested Struct, whereas the title member is a scalar member.

Figure: Struct Variables Referencing Members of a Struct



All these variables point to the same Struct in memory.

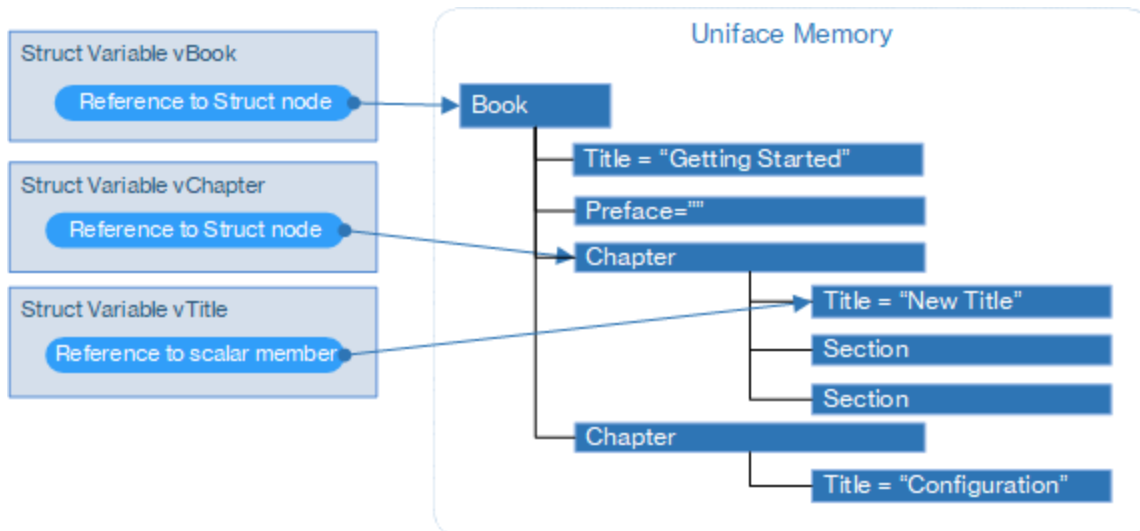
- A change made through one **struct** variable, is reflected in other variables that refer to the same Struct or Struct member. For example, changing the title of the first chapter (vChapter):

```
vChapter->Title = "New Title"
```

means that the following expressions all evaluate to "New Title".

```
$1 = vTitle
$2 = vChapter->Title
$3 = vBook->chapter{1}->Title
; $1 == $2 == $3 = "New Title"
```

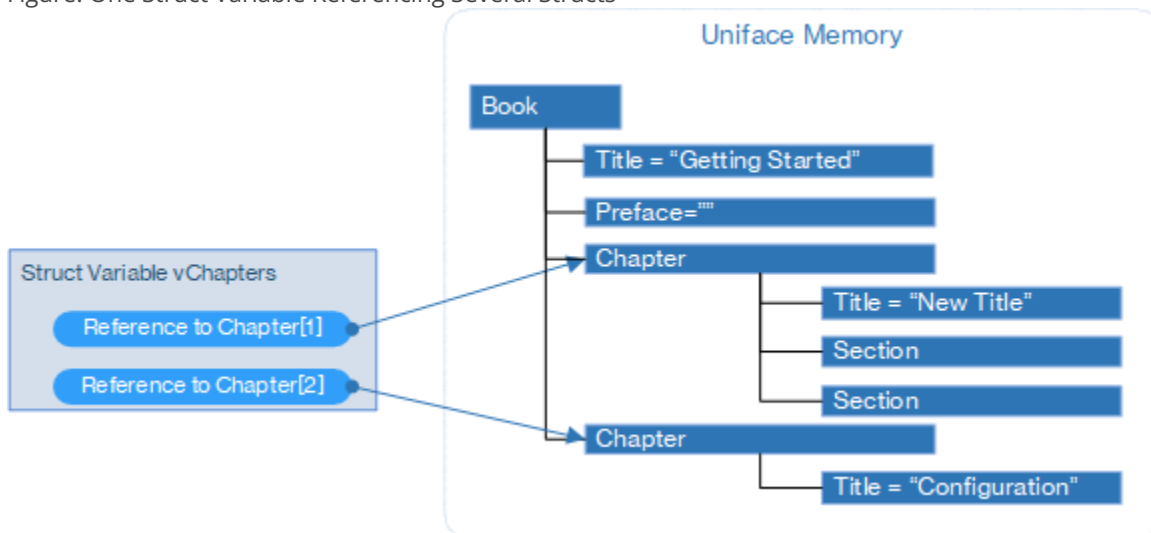
Figure: Changes Made to Struct Are Visible to All Variables That Point To It



- A single Struct variable can refer to multiple Structs, which makes it possible to manipulate a group of Structs, instead of handling them individually. For example, each chapter member of the vBook is itself a nested Struct. The following code results in vStruct containing a collection of Structs named chapter:

```
vChapters = vBook->chapter
```

Figure: One Struct Variable Referencing Several Structs



- A **struct** variable that contains a collection of only one reference to a Struct node can be handled as a single Struct rather than a collection. This means, for example, that you do not have to specify an index when accessing it. For example, there is only one Preface member in a book Struct, so you can use:

```
vBook->preface
```

instead of using the Struct index operator to specify a particular reference in the collection:

```
vStruct{1}
```

- A **struct** variable can refer to zero Structs. In this case, the collection size (**\$collsize**) is 0.
- A **struct** variable or expression whose value is NULL is interpreted as being a collection of references to zero Structs. For more information, see [Null Values](#).



- The term **struct** variables also encompasses **struct** parameters and non-database fields. For more information, see [Passing Struct Parameters](#).
- ProcScript variables and parameters of type **any** can also be used to refer to Structs, because Uniface implicitly converts the data type.

```
variables
  any vAny
  struct vStruct
endvariables
vStruct = $newstruct ; Create a Struct
vStruct->title = "Installation Guide" ; Add a member
vAny = vStruct ; Assign vStruct to vAny
```

vAny now points to same Struct as vStruct.


# Struct Access Operators

You can address collections of Structs by name and individual Structs by name and index using the Struct access operators—the *de-reference* operator (->) and the index operator ({N}).

A variable or parameter of type **struct** is an ordered collection of references to one or more Structs. A collection of one can be treated as an individual Struct.

Each Struct in the collection may or may not have a name, and may include multiple members with the same name. You therefore need to be able to access Structs by name (if they have one) or by their index position in the Struct (if they do not have a name, or there are multiple members with the same name).

**Table: Struct Access Operators**

Operator	Symbol	Syntax	Description
De-reference operator	->	Variable-> <i>Name</i>	Returns a collection of references to all Struct members with the specified name.
	->*	<i>Variable</i> ->*	When used with the * asterisk wildcard, it returns a collection of references to all members of a Struct. <div>  <b>Important:</b> The * wildcard can only be used after the de-reference operator. It cannot be used in constructs such as vStruct-&gt;c* to get all Structs whose names begin with c.           </div>
Struct index operator	{ N }	<i>Variable</i> -> <i>Name</i> { <i>N</i> }	Returns a reference to a single member of a Struct based on its index position in the collection of references.

# Example: Access Structs by Name or Index

The Struct access operators enable you to:

- Access members by name; for example, get all Struct members named chapter:

```
vBookStruct->chapter
```

- Access members by name and index; for example, get the second member named chapter.

```
vBookStruct->chapter{2}
```

- Get a collection of Structs; for example, get all members of the Struct referred to by variable vBookStruct:

```
vBookStruct->*
```

This returns the title, toc, preface, and all chapter members.

- Get all members in several Structs; for example, get all members of all Struct nodes called chapter:

```
vBookStruct->chapter->*
```

This returns title and section members.

- Access a Struct from a collection by its index position only; for example, get the second member in a Struct, regardless of its name:

```
vBookStruct->*{2}
```

## Related concepts

[Operators](#)

[struct](#)

## Struct Dereference Operator (->)

Access the members of a Struct by name.

*StructVariable -> MemberName | StructFunction*

## Arguments

- *StructVariable*—variable or parameter of type **struct** which refers to one or more Structs
- *MemberName*—Struct member identifier; can be a literal name or a string; for example `mem` or `"mem"`. An empty string `""` returns Structs with no name.
- *Function*—Struct function. For more information, see [ProcScript: Struct Functions](#).

## Return Values

Returns an ordered collection of references to members of the Struct to which *StructVariable* refers.

If *StructVariable* refers to a collection of Structs rather than to a single Struct, the expression is evaluated for each Struct in the collection, and the resulting collections are combined into a single ordered collection of Structs.

If there are no members that satisfy the criteria, the result is NULL and an error is set in `$procerror`.

**Table: Values of `$procerror` Commonly Returned Following `->`**

Error number	Error Constant	Meaning
-84	ACTERR_NO_OBJECT	Struct operator applied to non-Struct
-1153	USTRUCTERR_INDEX_NOT_ALLOWED	Struct index is not allowed
-1155	USTRUCTERR_MEMBER_NOT_FOUND	The Struct does not contain a member with the specified name.
-1157	USTRUCTERR_ILLEGAL_MEMBER_TYPE	<i>StructVariable</i> is not of type <b>struct</b> or <b>any</b>

## Use


Allowed in all components.

## Description

The arrow `->` is known as the dereference operator because the Struct being referred to changes (is de-referenced) from the one on the left to the one on the right. The dereference operator can be used multiple times in a statement to travers the nested levels of a Struct to get to the Struct member you need. For example, if you have a Struct referring to book that includes a Preface, you need to start from the top of the Struct nad use the dereference operator to get the Preface, and use it again to get the Acknowledgements section:

```
vStruct = Book->Preface->Acknowledgements
```

You can use the dereference operator to address Structs by name. By specifying an empty string (`""`), you can access nameless Structs. By using the `*` asterisk wildcard after the operator, you can get a collection of references to all members of a Struct.

 **Important:** The `*` wildcard can only be used after the dereference operator. It cannot be used in constructs such as `vStruct->c*` to get all Structs whose names begin with `c.` `->*` is actually a separate operator (the Struct collection operator).

## Member Names as Strings

*MemberName* can be specified as a string, which means that it is possible to:

- Use string substitution to access a field name. For example, the following code returns the members with the name as specified in field F1.

```
vStruct->"%F1%"
```

- Specify member names that include any character, including spaces, #, and \$.

```
vStruct->"first name"
```

- Address Struct members that have an empty string as name:

```
vStruct->"
```

If a Struct has several members with the same name, you can access one of these members specifically, by using the dereference operator in combination with the struct index operator {N}. For example:

```
vBookStruct->chapter{2}
```

## Related concepts

[Substitution in String Values](#)

## Struct Index Operator ({N})

Access a member of a Struct by its index position in the Struct.

- By index:

```
StructVariable -> * { Index }
```

- By name and index:

```
StructVariable -> Name { Index }
```

- With **struct** variable or parameter:

```
StructVariable { Index }
```

## Arguments

- *StructVariable*—variable or parameter of type **struct** which refers to one or Structs
- *Index*—index position of the member in the Struct; valid values are:
  - Integers > 0 and <= *N*, where *N* is the current number of members in the Struct
  - -1 for the last member in the Struct
  - *N*+1 when adding a member after the last member in the Struct.

## Return Values

Returns a reference to a single Struct member, or to NULL if there is no member at the specified index position

**Table: Possible Errors Returned in \$procerror after ->**

Error number	Error Constant	Meaning
-1155	USTRUCTERR_MEMBER_NOT_FOUND	The Struct does not contain a member with the specified name.
-1154	USTRUCTERR_INDEX_OUT_OF_RANGE	The Struct does not contain a member at the specified index position.
-1157	USTRUCTERR_ILLEGAL_MEMBER_TYPE	<i>StructVariable</i> is not a <b>struct</b> or <b>any</b> .

## Use

Allowed in all components.

## Description

You can use the index operator in combination with the dereference operator (->) to access a specific member in a collection of members with the same name. For example, you can extract the value of the second occurrence of the member with name `phone_number`, where multiple members have the same name:

```
vMobile = vStruct->phone_number{2}
```

When used with `->*`, you can access the member at a specific location in a Struct, regardless of the name. For example:

```
vTemp = vStruct->*[2]
```

## Struct Collection Operator (->\*)

Access all the members of a Struct.

*StructVariable* `->*`

## Arguments

*StructVariable*—variable or parameter of type **struct**, which refers to zero or more Structs

## Return Values

Returns an ordered collection of Structs.

**Table: Values of \$procerror Commonly Returned Following ->**

Error number	Error Constant	Meaning
-84	ACTERR_NO_OBJECT	Struct operator applied to non-Struct
-1153	USTRUCTERR_INDEX_NOT_ALLOWED	Struct index is not allowed
-1157	USTRUCTERR_ILLEGAL_MEMBER_TYPE	<i>StructVariable</i> is not a <b>struct</b> or <b>any</b>

## Use

Allowed in all components

## Description

The de-reference operator followed by a wildcard is actually a separate operator. No blanks are allowed between the arrow symbol and the asterisk.

The Struct Collection operator can be used to get a subset of members from a Struct. For example:

- `vStruct = vBook->{*}{3}`

`vStruct` refers to the third Struct member of `vBook`

- `vStruct = vBook->{*}{3}->section`

`vStruct` refers to a collection of members called `section`, belonging to the third member of `vBook`.

## Assigning by Value

You can use the collection operator in the left side of an assignment to insert new members in one or more Structs:

*StructVariable* ->{\*} = *Variable*

In this case:

1. All members are removed from the Struct.
2. The new members are inserted in the Struct.
  - If *Variable* refers to a Struct (or Structs), the new members copies of the individual Structs referred to by *Variable*.
  - If *Variable* is a scalar value, a single new member is inserted in each Struct referenced by *StructVariable*.

For more information, see [Adding, Copying, Moving, and Replacing Struct Members](#).

## Struct Functions

Struct functions enable you to get information about a Struct (such as the number of members it has), get or set Struct annotations, or perform actions such as inserting and moving Struct members.

Syntactically, they are treated as Struct members and are accessed using the dereference operator (->):

*Struct -> StructFunction*

For example:

```
vBook->$membercount
```

Unlike true members, Struct functions are not part of the Struct member list and they do not have an index.

Struct function names always begin with a dollar sign \$. Unlike other functions, they do not use parentheses ().

**Table: Struct Functions**

ProcScript	Description
<a href="#">\$collSize</a>	Get the number of Structs in the collection.
<a href="#">\$dbgString</a>	Get a string that represents the Struct or Struct collection.
<a href="#">\$dbgStringPlain</a>	Get a string that represents the Struct or Struct collection, but without annotations.
<a href="#">\$index</a>	Get or set the index of the Struct in a Struct collection.
<a href="#">\$isLeaf</a>	Check whether a Struct is a Struct leaf (the logical end point in a Struct tree).
<a href="#">\$isScalar</a>	Check whether a Struct is a scalar Struct.
<a href="#">\$istags</a>	Check whether the Struct is a <b>\$tags</b> Struct for another Struct.
<a href="#">\$memberCount</a>	Get the number of members in a Struct.
<a href="#">\$name</a>	Get the name of a Struct.
<a href="#">\$parent</a>	Get or set the parent of the Struct.
<a href="#">\$scalar</a>	Get or set the scalar members of a Struct.
<a href="#">\$tags</a>	Get or set annotations for a Struct.

## Setting Struct Functions

Although most Struct functions only return information, the **\$index**, **\$parent**, and **\$tag** functions can be used to change data. You can use:

- **\$index** to insert a member into a Struct at the specified index.
- **\$parent** to move a Struct member from one Struct to another.
- **\$tags** to set the value of annotations, or define your own annotations for use in conversion routines.

For more information, see [Struct Annotations](#).

### Related reference

[ProcScript: Struct Functions](#)