

# The lt3rawobjects package

Paolo De Donato

Released on 2022/12/27 Version 2.3-beta

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Objects and proxies</b>	<b>2</b>
<b>3</b>	<b>Put objects inside objects</b>	<b>3</b>
3.1	Put a pointer variable . . . . .	3
3.2	Clone the inner structure . . . . .	4
3.3	Embedded objects . . . . .	5
<b>4</b>	<b>Minimal objects</b>	<b>6</b>
<b>5</b>	<b>Library functions</b>	<b>6</b>
5.1	Base object functions . . . . .	6
5.2	Members . . . . .	7
5.3	Methods . . . . .	9
5.4	Constant member creation . . . . .	10
5.5	Macros . . . . .	11
5.6	Proxy utilities and object creation . . . . .	11
<b>6</b>	<b>Examples</b>	<b>14</b>
<b>7</b>	<b>Templated proxies</b>	<b>16</b>
<b>8</b>	<b>Implementation</b>	<b>17</b>

## 1 Introduction

First to all notice that lt3rawobjects means “raw object(s)”, indeed lt3rawobjects introduces a new mechanism to create objects like the well known C structures. The functions exported by this package are quite low level, and many important mechanisms like member protection and name resolution aren’t already defined and should be introduced by intermediate packages. Higher level libraries built on top of lt3rawobjects could also implement an improved and simplified syntax since the main focus of lt3rawobjects is versatility and expandability rather than common usage.

This packages follows the [SemVer](https://semver.org/) specification (<https://semver.org/>). In particular any major version update (for example from 1.2 to 2.0) may introduce incompatible

changes and so it's not advisable to work with different packages that require different major versions of `lt3rawobjects`. Instead changes introduced in minor and patch version updates are always backward compatible, and any withdrawn function is declared deprecated instead of being removed.

## 2 Objects and proxies

Usually an object in programming languages can be seen as a collection of variables (organized in different ways depending on the chosen language) treated as part of a single entity. In `lt3rawobjects` objects are collections of

- `LATEX3` variables, called *members*;
- `LATEX3` functions, called *methods*;
- generic control sequences, called simply *macros*;
- other *embedded objects*.

Both members and methods can be retrieved from a string representing the container object, that is the *address* of the object and act like the address of a structure in C.

An address is composed of two parts: the *module* in which variables are created and an *identifier* that identify uniquely the object inside its module. It's up to the caller that two different objects have different identifiers. The address of an object can be obtained with the `\object_address` function. Identifiers and module names should not contain numbers, `#`, `:` and `_` characters in order to avoid conflicts with hidden auxiliary commands. However you can use non letter characters like `-` in order to organize your members and methods.

Moreover normal control sequences have an address too, but it's simply any token list for which a `c` expansion retrieves the original control sequence. We impose also that any `x` or `e` fully expansion will be a string representing the control sequence's name, for this reason inside an address `#` characters and `\exp_not` functions aren't allowed.

In `lt3rawobjects` objects are created from an existing object that have a suitable inner structure. These objects that can be used to create other objects are called *proxy*. Every object is generated from a particular proxy object, called *generator*, and new objects can be created from a specified proxy with the `\object_create` functions.

Since proxies are themselves objects we need a proxy to instantiate user defined proxies, you can use the `proxy` object in the `rawobjects` module to create you own proxy, which address is held by the `\c_proxy_address_str` variable. Proxies must be created from the `proxy` object otherwise they won't be recognized as proxies. Instead of using `\object_create` to create proxies you can directly use the function `\proxy_create`.

Each member or method inside an object belongs to one of these categories:

1. *mutables*;
2. *near constants*;
3. *remote constants*.

**Warning:** Currently only members (variables) can be mutables, not methods. Mutable members can be added in future releases if they'll be needed.

Members declared as mutables works as normal variables: you can modify their value and retrieve it at any time. Instead members and methods declared as near constant works as constants: when you create them you must specify their initial value (or function body for methods) and you won't be allowed to modify it later. Remote constants for an object are simply near constants defined in its generator: all near constants defined inside a proxy are automatically visible as remote constants to every object generated from that proxy. Usually functions involving near constants have `nc` inside their name, and `rc` if instead they use remote constants.

Instead of creating embedded objects or mutable members in each of your objects you can push their specifications inside the generating proxy via `\proxy_push_embedded`, `\proxy_push_member`. In this way either object created from such proxy will have the specified members and embedded objects. Specify mutable members in this way allows you to omit that member type in some functions as `\object_member_adr` for example, their member type will be deduced automatically from its specification inside generating proxy.

Objects can be declared public, private and local, global. In a public/private object every nonconstant member and method is declared public/private, but inside local/global object only assignation to mutable members is performed locally/globally since allocation is always performed globally via `\(type)_new:Nn` functions (nevertheless members will be accordingly declared `g_` or `l_`). This is intentional in order to follow the L<sup>A</sup>T<sub>E</sub>X3 guidelines about variables managment, for additional motivations you can see [this thread](#) in the L<sup>A</sup>T<sub>E</sub>X3 repository.

Address of members/methods can be obtained with functions in the form `\object_<item><category>_adr` where `<item>` is `member`, `method`, `macro` or `embedded` and `<category>` is `nc` for near constants, `rc` for remote ones and empty for others. For example `\object_rcmethod_adr` retrieves the address of specified remote constant method.

## 3 Put objects inside objects

Sometimes it's necessary to include other objects inside an object, and since objects are structured data types you can't put them directly inside a variable. However `lt3rawobjects` provides some workarounds that allows you to include objects inside other objects, each with its own advantages and disadvantages.

In the following examples we're in module `mymod` and we want to put inside object `A` another object created with proxy `prx`.

### 3.1 Put a pointer variable

A simple solution is creating that object outside `A` with `\object_create`

```
\object_create:nnnNN
{ \object_address:nn{ mymod }{ prx } }{ mymod }{ B } ....
```

and then creating a pointer variable inside `A` (usually of type `t1` or `str`) holding the newly created address:

```
\object_new_member:nnn
{
```

```

        \object_address:nn{ mymod }{ A }
    }{ pointer }{ t1 }

\tl_(g)set:cn
{
    \object_new_member:nnn
    {
        \object_address:nn{ mymod }{ A }
    }{ pointer }{ t1 }
}
{
    \object_address:nn{ mymod }{ B }
}

```

you can access the pointed object by calling `\object_member_use` on `pointer` member.

### Advantages

- Simple and no additional function needed to create and manage included objects;
- you can share the same object between different containers;
- included objects are objects too, you can use address stored in pointer member just like any object address.

### Disadvantages

- You must manually create both the objects and link them;
- creating objects also creates additional hidden variables, taking so (little) additional space.

## 3.2 Clone the inner structure

Instead of referring a complete object you can just clone the inner structure of `prx` and put inside `A`. For example if `prx` declares member `x` of type `str` and member `y` of type `int` then you can do

```

\object_new_member:nnn
{
    \object_address:nn{ mymod }{ A }
} { prx-x }{ str }
\object_new_member:nnn
{
    \object_address:nn{ mymod }{ A }
} { prx-y }{ int }

```

and then use `prx-x`, `prx-y` as normal members of `A`.

### Advantages

- Simple and no additional function needed to create and manage included objects;
- you can put these specifications inside a proxy so that every object created with it will have the required members/methods;
- no hidden variable created, lowest overhead among the proposed solutions.

### Disadvantages

- Cloning the inner structure doesn't create any object, so you don't have any object address nor you can share the included "object" unless you share the container object too.

## 3.3 Embedded objects

From `lt3rawobjects 2.2` you can put *embedded objects* inside objects. Embedded objects are created with `\embedded_create` function

```
\embedded_create:nnn
{
  \object_address:nn{ mymod }{ A }
}{ prx }{ B }
```

and addresses of embedded objects can be retrieved with function `\object_embedded_adr`. You can also put the definition of embedded objects in a proxy by using `\proxy_push_embedded` just like `\proxy_push_member`.

### Advantages

- You can put a declaration inside a proxy so that embedded objects are automatically created during creation of parent object;
- included objects are objects too, you can use address stored in pointer member just like any object address.

### Disadvantages

- Needs additional functions available for version 2.2 or later;
- embedded objects must have the same scope and visibility of parent one;
- creating objects also creates additional hidden variables, taking so (little) additional space.

## 4 Minimal objects

When you create an object with a proxy additional hidden constant are added to it in order to hold information about the object. Some held data are

- the module in which your object is declared;
- the proxy that created it;
- its scope and visibility.

These hidden variable are very useful and are needed to manage remote constants and access members and methods without specify their type, scope and visibility each time. Near constants and macros are perhaps the only items that doesn't need these internal information.

Sometimes you don't want to create such internal variables for your objects despite objects without these hidden fields don't work well with third part libraries. Object with some of all these fields missing are called *minimal objects*.

You don't need a special function to instantiate a minimal object: any valid object address that has not been used could be used as an empty minimal object which you can populate it with subsequential calls to `\object_new_member`, `\object_newconst` or any other function (remember that for members you should specify every time their scope and visibility).

If you really really need a proxy that creates minimal objects from version 2.3 you can use the `proxymin` proxy in `rawobjects` module. Also if you want to create only scope and visibility fields in your objects (in order to avoid specifying them every time) you can use instead the `proxylgt` proxy.

## 5 Library functions

### 5.1 Base object functions

---

<code>\object_address:nn</code> *	<code>\object_address:nn {&lt;module&gt;} {&lt;id&gt;}</code>
-----------------------------------	---

---

Composes the address of object in module `<module>` with identifier `<id>` and places it in the input stream. Notice that `<module>` and `<id>` are converted to strings before composing them in the address, so they shouldn't contain any command inside. If you want to execute its content you should use a new variant, for example `V`, `f` or `e` variants.

From: 1.0

---

<code>\object_address_set:Nnn</code>	<code>\object_address_set:nn &lt;str var&gt; {&lt;module&gt;} {&lt;id&gt;}</code>
<code>\object_address_gset:Nnn</code>	

---

Stores the adress of selected object inside the string variable `<str var>`.

From: 1.1

---

<code>\object_embedded_adr:nn</code> *	<code>\object_embedded_adr:nn {&lt;address&gt;} {&lt;id&gt;}</code>
<code>\object_embedded_adr:Vn</code> *	

---

Compose the address of embedded object with name `<id>` inside the parent object with address `<address>`. Since an embedded object is also an object you can use this function for any function that accepts object addresses as an argument.

From: 2.2

<hr/>	
\object_if_exist_p:n *	\object_if_exist_p:n {<address>}
\object_if_exist_p:V *	\object_if_exist:nTF {<address>} {<true code>} {<false code>}
\object_if_exist:nTF *	Tests if an object was instantiated at the specified address.
\object_if_exist:VTF *	From: 1.0
<hr/>	
\object_get_module:n *	\object_get_module:n {<address>}
\object_get_module:V *	\object_get_proxy_adr:n {<address>}
\object_get_proxy_adr:n *	Get the object module and its generator.
\object_get_proxy_adr:V *	From: 1.0
<hr/>	
\object_if_local_p:n *	\object_if_local_p:n {<address>}
\object_if_local_p:V *	\object_if_local:nTF {<address>} {<true code>} {<false code>}
\object_if_local:nTF *	Tests if the object is local or global.
\object_if_local:VTF *	From: 1.0
\object_if_global_p:n *	
\object_if_global_p:V *	
\object_if_global:nTF *	
\object_if_global:VTF *	
<hr/>	
\object_if_public_p:n *	\object_if_public_p:n {<address>}
\object_if_public_p:V *	\object_if_public:nTF {<address>} {<true code>} {<false code>}
\object_if_public:nTF *	Tests if the object is public or private.
\object_if_public:VTF *	From: 1.0
\object_if_private_p:n *	
\object_if_private_p:V *	
\object_if_private:nTF *	
\object_if_private:VTF *	
<hr/>	

## 5.2 Members

<hr/>	
\object_member_adr:nnn *	\object_member_adr:nnn {<address>} {<member name>} {<member type>}
\object_member_adr:(Vnn nnv) *	\object_member_adr:nn {<address>} {<member name>}
\object_member_adr:nn *	
\object_member_adr:Vn *	
<hr/>	

Fully expands to the address of specified member variable. If type is not specified it'll be retrieved from the generator proxy, but only if member is specified in the generator.

From: 1.0

<hr/>	
\object_member_adr:nnnNN *	\object_member_adr:nnnNN {<address>} {<member name>} {<member type>}
\object_member_adr:(VnnNN nnncc) *	<scope> <visibility>
<hr/>	

Same as \object\_member\_adr but scope and visibility are specified as arguments instead of reading hidden variables. This is useful for objects created without an internal auxiliary structure.

From: 2.3

---

<code>\object_member_if_exist_p:nnn</code>	<code>*</code>	<code>\object_member_if_exist_p:nnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;}</code>
<code>\object_member_if_exist_p:Vnn</code>	<code>*</code>	<code>type)}</code>
<code>\object_member_if_exist:nnnTF</code>	<code>*</code>	<code>\object_member_if_exist:nnnTF {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
<code>\object_member_if_exist:VnnTF</code>	<code>*</code>	<code>type)}</code>
<code>\object_member_if_exist_p:nn</code>	<code>*</code>	<code>\object_member_if_exist_p:nn {&lt;address&gt;} {&lt;member name&gt;}</code>
<code>\object_member_if_exist_p:Vn</code>	<code>*</code>	<code>\object_member_if_exist:nnTF {&lt;address&gt;} {&lt;member name&gt;} {&lt;true code&gt;}</code>
<code>\object_member_if_exist:nnTF</code>	<code>*</code>	<code>{&lt;false code&gt;}</code>
<code>\object_member_if_exist:VnTF</code>	<code>*</code>	

---

Tests if the specified member exist.

From: 2.0

---

<code>\object_member_type:nn</code>	<code>*</code>	<code>\object_member_type:nn {&lt;address&gt;} {&lt;member name&gt;}</code>
<code>\object_member_type:Vn</code>	<code>*</code>	

---

Fully expands to the type of member *<member name>*. Use this function only with member variables specified in the generator proxy, not with other member variables.

From: 1.0

---

<code>\object_new_member:nnn</code>	<code>\object_new_member:nnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;}</code>
<code>\object_new_member:(Vnn nnv)</code>	

---

Creates a new member variable with specified name and type. You can't retrieve the type of these variables with `\object_member_type` functions.

From: 1.0

---

<code>\object_new_member:nnnNN</code>	<code>\object_new_member:nnnNN {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;} &lt;scope&gt;</code>
<code>\object_new_member:VnnNN</code>	<code>&lt;visibility&gt;</code>

---

Same as `\object_new_member:nnn` but with specified scope and visibility.

From: 2.3

---

<code>\object_member_use:nnn</code>	<code>*</code>	<code>\object_member_use:nnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;}</code>
<code>\object_member_use:(Vnn nnv)</code>	<code>*</code>	<code>\object_member_use:nn {&lt;address&gt;} {&lt;member name&gt;}</code>
<code>\object_member_use:nn</code>	<code>*</code>	
<code>\object_member_use:Vn</code>	<code>*</code>	

---

Uses the specified member variable.

From: 1.0

---

<code>\object_member_use:nnnNN</code>	<code>*</code>	<code>\object_member_use:nnnNN {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;}</code>
<code>\object_member_use:(VnnNN nnncc)</code>	<code>*</code>	<code>&lt;scope&gt; &lt;visibility&gt;</code>

---

Same as `\object_member_use:nnn` but with the specified scope and visibility.

From: 2.3

---

<code>\object_member_set:nnnn</code>	<code>\object_member_set:nnnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;}</code>
<code>\object_member_set:(nnvn Vnnn)</code>	<code>{&lt;value&gt;}</code>
<code>\object_member_set:nnn</code>	<code>\object_member_set:nnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;value&gt;}</code>
<code>\object_member_set:Vnn</code>	

---

Sets the value of specified member to *<value>*. It calls implicitly `\<member type>_(g)set:cn` then be sure to define it before calling this method.

From: 2.1



---

<code>\object_member_set:nnnNNn</code>	<code>\object_member_set:nnnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;}</code>
<code>\object_member_set:(VnnNNn nnnccn)</code>	<code>&lt;scope&gt; &lt;visibility&gt; {&lt;value&gt;}</code>

---

Same as `\object_member_set:nnnn` but with specified scope and visibility.  
From: 2.3

---

<code>\object_member_set_eq:nnnN</code>	<code>\object_member_set_eq:nnnN {&lt;address&gt;} {&lt;member name&gt;}</code>
<code>\object_member_set_eq:(nnvN VnnN nnnc Vnnnc)</code>	<code>{&lt;member type&gt;} &lt;variable&gt;</code>
<code>\object_member_set_eq:nnN</code>	<code>\object_member_set_eq:nnN {&lt;address&gt;} {&lt;member name&gt;}</code>
<code>\object_member_set_eq:(VnN nnnc Vnc)</code>	<code>&lt;variable&gt;</code>

---

Sets the value of specified member equal to the value of *<variable>*.  
From: 1.0

---

<code>\object_ncmember_adr:nnn</code>	<code>*</code>	<code>\object_ncmember_adr:nnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;}</code>
<code>\object_ncmember_adr:(Vnn vnn)</code>	<code>*</code>	
<code>\object_rcmember_adr:nnn</code>	<code>*</code>	
<code>\object_rcmember_adr:Vnn</code>	<code>*</code>	

---

Fully expands to the address of specified near/remote constant member.  
From: 2.0

---

<code>\object_ncmember_if_exist_p:nnn</code>	<code>*</code>	<code>\object_ncmember_if_exist_p:nnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;member</code>
<code>\object_ncmember_if_exist_p:Vnn</code>	<code>*</code>	<code>type}&gt;</code>
<code>\object_ncmember_if_exist:nnnTF</code>	<code>*</code>	<code>\object_ncmember_if_exist:nnnTF {&lt;address&gt;} {&lt;member name&gt;} {&lt;member</code>
<code>\object_ncmember_if_exist:VnnTF</code>	<code>*</code>	<code>type}&gt; {&lt;true code&gt;} {&lt;false code&gt;}</code>
<code>\object_rcmember_if_exist_p:nnn</code>	<code>*</code>	
<code>\object_rcmember_if_exist_p:Vnn</code>	<code>*</code>	
<code>\object_rcmember_if_exist:nnnTF</code>	<code>*</code>	
<code>\object_rcmember_if_exist:VnnTF</code>	<code>*</code>	

---

Tests if the specified member constant exist.  
From: 2.0

---

<code>\object_ncmember_use:nnn</code>	<code>*</code>	<code>\object_ncmember_use:nnn {&lt;address&gt;} {&lt;member name&gt;} {&lt;member type&gt;}</code>
<code>\object_ncmember_use:Vnn</code>	<code>*</code>	
<code>\object_rcmember_use:nnn</code>	<code>*</code>	Uses the specified near/remote constant member.
<code>\object_rcmember_use:Vnn</code>	<code>*</code>	From: 2.0

---

### 5.3 Methods

Currentlu only constant methods (near and remote) are implemented in `lt3rawobjects` as explained before.

---

<code>\object_ncmethod_adr:nnn</code>	<code>*</code>	<code>\object_ncmethod_adr:nnn {&lt;address&gt;} {&lt;method name&gt;} {&lt;method</code>
<code>\object_ncmethod_adr:(Vnn vnn)</code>	<code>*</code>	<code>variant}&gt;</code>
<code>\object_rcmethod_adr:nnn</code>	<code>*</code>	
<code>\object_rcmethod_adr:Vnn</code>	<code>*</code>	

---

Fully expands to the address of the specified

- near constant method if `\object_ncmethod_adr` is used;
- remote constant method if `\object_rcmethod_adr` is used.

From: 2.0

---

<code>\object_ncmethod_if_exist_p:nnn</code>	<code>*</code>	<code>\object_ncmethod_if_exist_p:nnn</code>	<code>{\langle address \rangle}</code>	<code>{\langle method name \rangle}</code>	<code>{\langle method variant \rangle}</code>
<code>\object_ncmethod_if_exist_p:Vnn</code>	<code>*</code>				
<code>\object_ncmethod_if_exist:nnnTF</code>	<code>*</code>	<code>\object_ncmethod_if_exist:nnnTF</code>	<code>{\langle address \rangle}</code>	<code>{\langle method name \rangle}</code>	<code>{\langle method variant \rangle}</code>
<code>\object_ncmethod_if_exist:VnnTF</code>	<code>*</code>				
<code>\object_rcmethod_if_exist_p:nnn</code>	<code>*</code>				
<code>\object_rcmethod_if_exist_p:Vnn</code>	<code>*</code>				
<code>\object_rcmethod_if_exist:nnnTF</code>	<code>*</code>				
<code>\object_rcmethod_if_exist:VnnTF</code>	<code>*</code>				

---

Tests if the specified method constant exist.

From: 2.0

---

<code>\object_new_cmethod:nnnn</code>	<code>\object_new_cmethod:nnnn</code>	<code>{\langle address \rangle}</code>	<code>{\langle method name \rangle}</code>	<code>{\langle method arguments \rangle}</code>	<code>{\langle code \rangle}</code>
<code>\object_new_cmethod:Vnnn</code>					

---

Creates a new method with specified name and argument types. The `{\langle method arguments \rangle}` should be a string composed only by n and N characters that are passed to `\cs_new:Nn`.

From: 2.0

---

<code>\object_ncmethod_call:nnn</code>	<code>*</code>	<code>\object_ncmethod_call:nnn</code>	<code>{\langle address \rangle}</code>	<code>{\langle method name \rangle}</code>	<code>{\langle method variant \rangle}</code>
<code>\object_ncmethod_call:Vnn</code>	<code>*</code>				
<code>\object_rcmethod_call:nnn</code>	<code>*</code>				
<code>\object_rcmethod_call:Vnn</code>	<code>*</code>				

---

Calls the specified method. This function is expandable if and only if the specified method was not declared `protected`.

From: 2.0

## 5.4 Constant member creation

Unlike normal variables, constant variables in L<sup>A</sup>T<sub>E</sub>X3 are created in different ways depending on the specified type. So we dedicate a new section only to collect some of these functions readapted for near constants (remote constants are simply near constants created on the generator proxy).

---

<code>\object_newconst_tl:nnn</code>	<code>\object_newconst_⟨type⟩:nnn</code>	<code>{\langle address \rangle}</code>	<code>{\langle constant name \rangle}</code>	<code>{\langle value \rangle}</code>
<code>\object_newconst_tl:Vnn</code>				
<code>\object_newconst_str:nnn</code>				
<code>\object_newconst_str:Vnn</code>				
<code>\object_newconst_int:nnn</code>				
<code>\object_newconst_int:Vnn</code>				
<code>\object_newconst_clist:nnn</code>				
<code>\object_newconst_clist:Vnn</code>				
<code>\object_newconst_dim:nnn</code>				
<code>\object_newconst_dim:Vnn</code>				
<code>\object_newconst_skip:nnn</code>				
<code>\object_newconst_skip:Vnn</code>				
<code>\object_newconst_fp:nnn</code>				
<code>\object_newconst_fp:Vnn</code>				

---

Creates a constant variable with type `⟨type⟩` and sets its value to `⟨value⟩`.

From: 1.1

---

<code>\object_newconst_seq_from_clist:nnn</code>	<code>\object_newconst_seq_from_clist:nnn {&lt;address&gt;} {&lt;constant name&gt;}</code>
<code>\object_newconst_seq_from_clist:Vnn</code>	<code>{&lt;comma-list&gt;}</code>

---

Creates a `seq` constant which is set to contain all the items in `<comma-list>`.  
From: 1.1

---

<code>\object_newconst_prop_from_keyval:nnn</code>	<code>\object_newconst_prop_from_keyval:nnn {&lt;address&gt;} {&lt;constant name&gt;}</code>
<code>\object_newconst_prop_from_keyval:Vnn</code>	<code>{</code> <code>  &lt;key&gt; = &lt;value&gt;, ...</code> <code>}</code>

---

Creates a `prop` constant which is set to contain all the specified key-value pairs.  
From: 1.1

---

<code>\object_newconst:nnnn</code>	<code>\object_newconst:nnnn {&lt;address&gt;} {&lt;constant name&gt;} {&lt;type&gt;} {&lt;value&gt;}</code>
------------------------------------	---

---

Expands to `\<type>_const:cn {<address>} {<value>}`, use it if you need to create simple constants with custom types.  
From: 2.1

## 5.5 Macros

---

<code>\object_macro_adr:nn *</code>	<code>\object_macro_adr:nn {&lt;address&gt;} {&lt;macro name&gt;}</code>
<code>\object_macro_adr:Vn *</code>	Address of specified macro.

---

From: 2.2

---

<code>\object_macro_use:nn *</code>	<code>\object_macro_use:nn {&lt;address&gt;} {&lt;macro name&gt;}</code>
<code>\object_macro_use:Vn *</code>	Uses the specified macro. This function is expandable if and only if the specified macro is it.

---

From: 2.2

There isn't any standard function to create macros, and macro declarations can't be inserted in a `proxy` object. In fact a macro is just an unspecialized control sequence at the disposal of users that usually already know how to implement them.

## 5.6 Proxy utilities and object creation

---

<code>\object_if_proxy_p:n *</code>	<code>\object_if_proxy_p:n {&lt;address&gt;}</code>
<code>\object_if_proxy_p:V *</code>	<code>\object_if_proxy:nTF {&lt;address&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
<code>\object_if_proxy:nTF *</code>	Test if the specified object is a proxy object.
<code>\object_if_proxy:VTF *</code>	From: 1.0

---

<hr/>	
<code>\object_test_proxy_p:nn</code> *	<code>\object_test_proxy_p:nn {⟨object address⟩} {⟨proxy address⟩}</code>
<code>\object_test_proxy_p:Vn</code> *	<code>\object_test_proxy:nnTF {⟨object address⟩} {⟨proxy address⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_test_proxy:nnTF</code> *	
<code>\object_test_proxy:VnTF</code> *	Test if the specified object is generated by the selected proxy, where <i>⟨proxy variable⟩</i> is a string variable holding the proxy address.

**TeXhackers note:** Remember that this command uses internally an `e` expansion so in older engines (any different from Lua<sup>A</sup>TeX before 2019) it'll require slow processing. Don't use it in speed critical parts, instead use `\object_test_proxy:nN`.

From: 2.0

<hr/>	
<code>\object_test_proxy_p:nN</code> *	<code>\object_test_proxy_p:nN {⟨object address⟩} ⟨proxy variable⟩</code>
<code>\object_test_proxy_p:VN</code> *	<code>\object_test_proxy:nNTF {⟨object address⟩} ⟨proxy variable⟩ {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_test_proxy:nNTF</code> *	
<code>\object_test_proxy:VNTF</code> *	Test if the specified object is generated by the selected proxy, where <i>⟨proxy variable⟩</i> is a string variable holding the proxy address. The <code>:nN</code> variant don't use <code>e</code> expansion, instead of <code>:nn</code> command, so it can be safely used with older compilers.

From: 2.0

<hr/>	
<code>\c_proxy_address_str</code>	The address of the <code>proxy</code> object in the <code>rawobjects</code> module.

From: 1.0

<hr/>	
<code>\c_proxy_min_adr_str</code>	The address of the <code>proxymn</code> object in the <code>rawobjects</code> module.

From: 2.3

<hr/>	
<code>\c_proxy_lgt_adr_str</code>	The address of the <code>proxylgt</code> object in the <code>rawobjects</code> module.

From: 2.3

<hr/>	
<code>\c_proxy_empty_adr_str</code>	The address of the <code>empty proxy</code> object in the <code>rawobjects</code> module. It can be used to create empty objects.

From: 2.3

<hr/>	
<code>\object_create:nnnNN</code>	<code>\object_create:nnnNN {⟨proxy address⟩} {⟨module⟩} {⟨id⟩} {⟨scope⟩} {⟨visibility⟩}</code>
<code>\object_create:VnnNN</code>	Creates an object by using the proxy at <i>⟨proxy address⟩</i> and the specified parameters.

From: 1.0

<hr/>	
<code>\embedded_create:nnn</code>	<code>\embedded_create:nnn {⟨parent object⟩} {⟨proxy address⟩} {⟨id⟩}</code>
<code>\embedded_create:(Vnn nvn)</code>	Creates an embedded object with name <i>⟨id⟩</i> inside <i>⟨parent object⟩</i> .

From: 2.2

<hr/>	
<code>\embedded_create:nnnnNN</code>	<code>\embedded_create:nnnnNN {⟨parent object⟩} {⟨proxy address⟩} {⟨module⟩} {⟨id⟩}</code>
<code>\embedded_create:nnvncc</code>	<code>{⟨scope⟩} {⟨visibility⟩}</code>

Same as `\embedded_create:nnn` but with the specified arguments. Use it only if *⟨parent object⟩* doesn't provide information about *⟨module⟩*, *⟨scope⟩* or *⟨visibility⟩*.

From: 2.3

<hr/>	
\c_object_local_str	Possible values for <i>&lt;scope&gt;</i> parameter.
\c_object_global_str	From: 1.0
<hr/>	
\c_object_public_str	Possible values for <i>&lt;visibility&gt;</i> parameter.
\c_object_private_str	From: 1.0
<hr/>	
\object_create_set:NnnnNN	\object_create_set:NnnnNN <i>&lt;str var&gt;</i> <i>{&lt;proxy address&gt;}</i> <i>{&lt;module&gt;}</i>
\object_create_set:(NVnnNN NnnfNN)	<i>{&lt;id&gt;}</i> <i>&lt;scope&gt;</i> <i>&lt;visibility&gt;</i>
\object_create_gset:NnnnNN	
\object_create_gset:(NVnnNN NnnfNN)	
<hr/>	
	Creates an object and sets its fully expanded address inside <i>&lt;str var&gt;</i> .
	From: 1.0
<hr/>	
\object_allocate_incr:NNnnNN	\object_allocate_incr:NNnnNN <i>&lt;str var&gt;</i> <i>&lt;int var&gt;</i> <i>{&lt;proxy address&gt;}</i>
\object_allocate_incr:NNVnNN	<i>{&lt;module&gt;}</i> <i>&lt;scope&gt;</i> <i>&lt;visibility&gt;</i>
\object_gallocate_incr:NNnnNN	
\object_gallocate_incr:NNVnNN	
\object_allocate_gincr:NNnnNN	
\object_allocate_gincr:NNVnNN	
\object_gallocate_gincr:NNnnNN	
\object_gallocate_gincr:NNVnNN	
<hr/>	
	Build a new object address with module <i>&lt;module&gt;</i> and an identifier generated from <i>&lt;proxy address&gt;</i> and the integer contained inside <i>&lt;int var&gt;</i> , then increments <i>&lt;int var&gt;</i> . This is very useful when you need to create a lot of objects, each of them on a different address. the <i>_incr</i> version increases <i>&lt;int var&gt;</i> locally whereas <i>_gincr</i> does it globally.
	From: 1.1
<hr/>	
\proxy_create:nnN	\proxy_create:nnN <i>{&lt;module&gt;}</i> <i>{&lt;id&gt;}</i> <i>&lt;visibility&gt;</i>
\proxy_create_set:NnnN	\proxy_create_set:NnnN <i>&lt;str var&gt;</i> <i>{&lt;module&gt;}</i> <i>{&lt;id&gt;}</i> <i>&lt;visibility&gt;</i>
\proxy_create_gset:NnnN	Creates a global proxy object.
<hr/>	
	From: 1.0
<hr/>	
\proxy_push_member:nnn	\proxy_push_member:nnn <i>{&lt;proxy address&gt;}</i> <i>{&lt;member name&gt;}</i> <i>{&lt;member type&gt;}</i>
\proxy_push_member:Vnn	Updates a proxy object with a new member specification, so that every subsequential object created with this proxy will have a member variable with the specified name and type that can be retrieved with <i>\object_member_type</i> functions.
<hr/>	
	From: 1.0
<hr/>	
\proxy_push_embedded:nnn	\proxy_push_embedded:nnn <i>{&lt;proxy address&gt;}</i> <i>{&lt;embedded object name&gt;}</i> <i>{&lt;embedded object proxy&gt;}</i>
\proxy_push_embedded:Vnn	Updates a proxy object with a new embedded object specification.
<hr/>	
	From: 2.2

---

```
\proxy_add_initializer:nN
\proxy_add_initializer:VN
```

---

```
\proxy_add_initializer:nN {<proxy address>} {<initializer>}
```

Pushes a new initializer that will be executed on each created objects. An initializer is a function that should accept five arguments in this order:

- the full expanded address of used proxy as an **n** argument;
- the module name as an **n** argument;
- the full expanded address of created object as an **n** argument;
- the scope as an **N** argument;
- the visibility as an **N** argument.

Initializer will be executed in the same order they're added.

---

```
\proxy_init_SV:nnnNN
\proxy_init_MP:nnnNN
```

---

```
\proxy_init_SV:nnnNN {<proxy address>} {<module>} {<address>} {<scope>} {<visibility>}
```

Initializers that can be added to proxies via `\proxy_add_initializer`. They respectively create scope, visibility and module, proxy internal variables in the newly created object ad `<address>`. They're both present in `proxy` and `empty` proxies and only `\proxy_init_SV` is present in `proxylgt`.

From: 2.3

---

```
\object_assign:nn
\object_assign:(Vn|nV|VV)
```

---

```
\object_assign:nn {<to address>} {<from address>}
```

Assigns the content of each variable of object at `<from address>` to each correlative variable in `<to address>`. Both the objects should be created with the same proxy object and only variables listed in the proxy are assigned.

From: 1.0

## 6 Examples

### Example 1

Create a public proxy with id `myproxy` with the specification of a single member variable with name `myvar` and type `tl`, then set its address inside `\l_myproxy_str`.

```
\str_new:N \l_myproxy_str
\proxy_create_set:NnnN \l_myproxy_str { example } { myproxy }
\c_object_public_str
\proxy_push_member:Vnn \l_myproxy_str { myvar } { tl }
```

Then create a new object with name `myobj` with that proxy, assign then token list `\c_dollar_str{}` to `myvar` and then print it.

```
\str_new:N \l_myobj_str
\object_create_set:NVnnNN \l_myobj_str \l_myproxy_str
{ example } { myobj } \c_object_local_str \c_object_public_str
\tl_set:cn
{
  \object_member_adr:Vn \l_myobj_str { myvar }
}
```

```
{ \c_dollar_str{} ~ dollar ~ \c_dollar_str{} }
\object_member_use:Vn \l_myobj_str { myvar }
```

Output: \$ dollar \$

If you don't want to specify an object identifier you can also do

```
\int_new:N \l_intc_int
\object_allocate_incr:NNVnNN \l_myobj_str \l_intc_int \l_myproxy_str
{ example } \c_object_local_str \c_object_public_str
\tl_set:cn
{
  \object_member_adr:Vn \l_myobj_str { myvar }
}
{ \c_dollar_str{} ~ dollar ~ \c_dollar_str{} }
\object_member_use:Vn \l_myobj_str { myvar }
```

Output: \$ dollar \$

## Example 2

In this example we create a proxy object with an embedded object inside.

Internal proxy

```
\proxy_create:nnN{ mymod }{ INT } \c_object_public_str
\proxy_push_member:nnn
{
  \object_address:nn{ mymod }{ INT }
}{ var }{ t1 }
```

Container proxy

```
\proxy_create:nnN{ mymod }{ EXT } \c_object_public_str
\proxy_push_embedded:nnn
{
  \object_address:nn{ mymod }{ EXT }
}
{ emb }
{
  \object_address:nn{ mymod }{ INT }
}
```

Now we create a new object from proxy EXT. It'll contain an embedded object created with INT proxy:

```
\str_new:N \g_EXTobj_str
\int_new:N \g_intcount_int
\object_gallocate_gincr:NNnnNN
\g_EXTobj_str \g_intcount_int
{
  \object_address:nn{ mymod }{ EXT }
}
{ mymod }
\c_object_local_str \c_object_public_str
```

and use the embedded object in the following way:

```
\object_member_set:nnn
{
  \object_embedded_adr:Vn \g_EXTObj_str { emb }
}{ var }{ Hi }
\object_member_use:nn
{
  \object_embedded_adr:Vn \g_EXTObj_str { emb }
}{ var }
```

Output: Hi

## 7 Templated proxies

At the current time there isn't a standardized approach to templated proxies. One problem of standardized templated proxies is how to define struct addresses for every kind of argument (token lists, strings, integer expressions, non expandable arguments, ...).

Even if there isn't currently a function to define every kind of templated proxy you can anyway define your templated proxy with your custom parameters. You simply need to define at least two functions:

- an expandable macro that, given all the needed arguments, fully expands to the address of your templated proxy. This address can be obtained by calling `\object_address {<module>} {<id>}` where `<id>` starts with the name of your templated proxy and is followed by a composition of specified arguments;
- a not expandable macro that tests if the templated proxy with specified arguments is instantiated and, if not, instantiate it with different calls to `\proxy_create` and `\proxy_push_member`.

In order to apply these concepts we'll provide a simple implementation of a linked list with a template parameter representing the type of variable that holds our data. A linked list is simply a sequence of nodes where each node contains your data and a pointer to the next node. For the moment we'll show a possible implementation of a template proxy class for such `node` objects.

First to all we define an expandable macro that fully expands to our node name:

```
\cs_new:Nn \node_address:n
{
  \object_address:nn { linklist }{ node - #1 }
}
```

where the `#1` argument is simply a string representing the type of data held by our linked list (for example `tl`, `str`, `int`, ...). Next we need a functions that instantiate our proxy address if it doesn't exist:

```
\cs_new_protected:Nn \node_instantiate:n
{
  \object_if_exist:nF {\node_address:n { #1 } }
  {
```



```

\proxy_create:nnN { linklist }{ node - #1 }
  \c_object_public_str
\proxy_push_member:nnn {\node_address:n { #1 } }
  { next }{ str }
\proxy_push_member:nnn {\node_address:n { #1 } }
  { data }{ #1 }
}
}

```

As you can see when `\node_instantiate` is called it first test if the proxy object exists. If not then it creates a new proxy with that name and populates it with the specifications of two members: a `next` member variable of type `str` that points to the next node, and a `data` member of the specified type that holds your data.

Clearly you can define new functions to work with such nodes, for example to test if the next node exists or not, to add and remove a node, search inside a linked list, ...

## 8 Implementation

```

1 <*package>
2 <@@=rawobjects>

\c_object_local_str
\c_object_global_str
\c_object_public_str
\c_object_private_str

3 \str_const:Nn \c_object_local_str {l}
4 \str_const:Nn \c_object_global_str {g}
5 \str_const:Nn \c_object_public_str {_}
6 \str_const:Nn \c_object_private_str {__}
7
8
9 \cs_new:Nn \__rawobjects_scope:N
10 {
11   \str_use:N #1
12 }
13
14 \cs_new:Nn \__rawobjects_scope_pfx:N
15 {
16   \str_if_eq:NNF #1 \c_object_local_str
17   { g }
18 }
19
20 \cs_new:Nn \__rawobjects_vis_var:N
21 {
22   \str_use:N #1
23 }
24
25 \cs_new:Nn \__rawobjects_vis_fun:N
26 {
27   \str_if_eq:NNT #1 \c_object_private_str
28   {
29     --
30   }
31 }
32

```

(End definition for `\c_object_local_str` and others. These variables are documented on page 13.)

`\object_address:nn` Get address of an object

```

33 \cs_new:Nn \object_address:nn {
34   \tl_to_str:n { #1 _ #2 }
35 }

```

(End definition for `\object_address:nn`. This function is documented on page 6.)

`\object_embedded_adr:nn` Address of embedded object

```

36
37 \cs_new:Nn \object_embedded_adr:nn
38 {
39   #1 \tl_to_str:n{ _SUB_ #2 }
40 }
41
42 \cs_generate_variant:Nn \object_embedded_adr:nn{ Vn }
43

```

(End definition for `\object_embedded_adr:nn`. This function is documented on page 6.)

`\object_address_set:Nnn` Saves the address of an object into a string variable

`\object_address_gset:Nnn`

```

44
45 \cs_new_protected:Nn \object_address_set:Nnn {
46   \str_set:Nn #1 { #2 _ #3 }
47 }
48
49 \cs_new_protected:Nn \object_address_gset:Nnn {
50   \str_gset:Nn #1 { #2 _ #3 }
51 }
52

```

(End definition for `\object_address_set:Nnn` and `\object_address_gset:Nnn`. These functions are documented on page 6.)

`\object_if_exist_p:n` Tests if object exists.

`\object_if_exist:nTF`

```

53
54 \prg_new_conditional:Nnn \object_if_exist:n { p, T, F, TF }
55 {
56   \cs_if_exist:cTF
57   {
58     \object_ncmember_adr:nnn
59     {
60       \object_embedded_adr:nn{ #1 }{ /_I_/ }
61     }
62     { S }{ str }
63   }
64   {
65     \prg_return_true:
66   }
67   {
68     \prg_return_false:
69   }
70 }
71

```

```

72 \prg_generate_conditional_variant:Nnn \object_if_exist:n { V }
73 { p, T, F, TF }
74

```

(End definition for \object\_if\_exist:nTF. This function is documented on page 7.)

**\object\_get\_module:n** Retrieve the name, module and generating proxy of an object  
**\object\_get\_proxy\_adr:n**

```

75 \cs_new:Nn \object_get_module:n {
76   \object_ncmember_use:nnn
77   {
78     \object_embedded_adr:nn{ #1 }{ /_I_/ }
79   }
80   { M }{ str }
81 }
82 \cs_new:Nn \object_get_proxy_adr:n {
83   \object_ncmember_use:nnn
84   {
85     \object_embedded_adr:nn{ #1 }{ /_I_/ }
86   }
87   { P }{ str }
88 }
89
90 \cs_generate_variant:Nn \object_get_module:n { V }
91 \cs_generate_variant:Nn \object_get_proxy_adr:n { V }

```

(End definition for \object\_get\_module:n and \object\_get\_proxy\_adr:n. These functions are documented on page 7.)

**\object\_if\_local\_p:n** Test the specified parameters.  
**\object\_if\_local:nTF**  
**\object\_if\_global\_p:n**  
**\object\_if\_global:nTF**  
**\object\_if\_public\_p:n**  
**\object\_if\_public:nTF**  
**\object\_if\_private\_p:n**  
**\object\_if\_private:nTF**

```

92 \prg_new_conditional:Nnn \object_if_local:n {p, T, F, TF}
93 {
94   \str_if_eq:cNTF
95   {
96     \object_ncmember_adr:nnn
97     {
98       \object_embedded_adr:nn{ #1 }{ /_I_/ }
99     }
100    { S }{ str }
101  }
102  \c_object_local_str
103  {
104    \prg_return_true:
105  }
106  {
107    \prg_return_false:
108  }
109 }
110
111 \prg_new_conditional:Nnn \object_if_global:n {p, T, F, TF}
112 {
113   \str_if_eq:cNTF
114   {
115     \object_ncmember_adr:nnn
116     {
117       \object_embedded_adr:nn{ #1 }{ /_I_/ }

```

```

118     }
119     { S }{ str }
120 }
121 \c_object_global_str
122 {
123     \prg_return_true:
124 }
125 {
126     \prg_return_false:
127 }
128 }
129
130 \prg_new_conditional:Nnn \object_if_public:n {p, T, F, TF}
131 {
132     \str_if_eq:cNTF
133     {
134         \object_ncmember_adr:nnn
135         {
136             \object_embedded_adr:nn{ #1 }{ /_I_/ }
137         }
138         { V }{ str }
139     }
140     \c_object_public_str
141     {
142         \prg_return_true:
143     }
144     {
145         \prg_return_false:
146     }
147 }
148
149 \prg_new_conditional:Nnn \object_if_private:n {p, T, F, TF}
150 {
151     \str_if_eq:cNTF
152     {
153         \object_ncmember_adr:nnn
154         {
155             \object_embedded_adr:nn{ #1 }{ /_I_/ }
156         }
157         { V }{ str }
158     }
159     \c_object_private_str
160     {
161         \prg_return_true:
162     }
163     {
164         \prg_return_false:
165     }
166 }
167
168 \prg_generate_conditional_variant:Nnn \object_if_local:n { V }
169 { p, T, F, TF }
170 \prg_generate_conditional_variant:Nnn \object_if_global:n { V }
171 { p, T, F, TF }

```

```

172 \prg_generate_conditional_variant:Nnn \object_if_public:n { V }
173 { p, T, F, TF }
174 \prg_generate_conditional_variant:Nnn \object_if_private:n { V }
175 { p, T, F, TF }

```

(End definition for `\object_if_local:nTF` and others. These functions are documented on page 7.)

`\object_macro_adr:nn` Generic macro address  
`\object_macro_use:nn`

```

176
177 \cs_new:Nn \object_macro_adr:nn
178 {
179   #1 \tl_to_str:n{ _MACRO_ #2 }
180 }
181
182 \cs_generate_variant:Nn \object_macro_adr:nn{ Vn }
183
184 \cs_new:Nn \object_macro_use:nn
185 {
186   \use:c
187   {
188     \object_macro_adr:nn{ #1 }{ #2 }
189   }
190 }
191
192 \cs_generate_variant:Nn \object_macro_use:nn{ Vn }
193

```

(End definition for `\object_macro_adr:nn` and `\object_macro_use:nn`. These functions are documented on page 11.)

`\object_member_adr:nnnNN` Macro address without object inference

```

194
195 \cs_new:Nn \object_member_adr:nnnNN
196 {
197   \__rawobjects_scope:N #4
198   \__rawobjects_vis_var:N #5
199   #1 \tl_to_str:n { _ MEMBER _ #2 _ #3 }
200 }
201
202 \cs_generate_variant:Nn \object_member_adr:nnnNN { VnnNN, nnncc }
203

```

(End definition for `\object_member_adr:nnnNN`. This function is documented on page 7.)

`\object_member_adr:nnn` Get the address of a member variable  
`\object_member_adr:nn`

```

204
205 \cs_new:Nn \object_member_adr:nnn
206 {
207   \object_member_adr:nnncc { #1 }{ #2 }{ #3 }
208   {
209     \object_ncmember_adr:nnn
210     {
211       \object_embedded_adr:nn{ #1 }{ /_I_/ }
212     }
213     { S }{ str }

```

```

214     }
215     {
216         \object_ncmember_adr:nnn
217         {
218             \object_embedded_adr:nn{ #1 }{ /_I_/ }
219         }
220         { V }{ str }
221     }
222 }
223
224 \cs_generate_variant:Nn \object_member_adr:nnn { Vnn, vnn, nnv }
225
226 \cs_new:Nn \object_member_adr:nn
227 {
228     \object_member_adr:nnv { #1 }{ #2 }
229     {
230         \object_rcmember_adr:nnn { #1 }
231         { #2 _ type }{ str }
232     }
233 }
234
235 \cs_generate_variant:Nn \object_member_adr:nn { Vn }
236

```

(End definition for `\object_member_adr:nnn` and `\object_member_adr:nn`. These functions are documented on page 7.)

**`\object_member_type:nn`** Deduce the member type from the generating proxy.

```

237
238 \cs_new:Nn \object_member_type:nn
239 {
240     \object_rcmember_use:nnn { #1 }
241     { #2 _ type }{ str }
242 }
243

```

(End definition for `\object_member_type:nn`. This function is documented on page 8.)

```

244
245 \msg_new:nnnn { rawobjects }{ noerr }{ Unspecified ~ scope }
246 {
247     Object ~ #1 ~ hasn't ~ a ~ scope ~ variable
248 }
249
250 \msg_new:nnnn { rawobjects }{ scoperr }{ Nonstandard ~ scope }
251 {
252     Operation ~ not ~ permitted ~ on ~ object ~ #1 ~
253     ~ since ~ it ~ wasn't ~ declared ~ local ~ or ~ global
254 }
255
256 \cs_new_protected:Nn \__rawobjects_force_scope:n
257 {
258     \cs_if_exist:cTF
259     {
260         \object_ncmember_adr:nnn

```

```

261         {
262             \object_embedded_adr:nn{ #1 }{ /_I_/ }
263         }
264         { S }{ str }
265     }
266     {
267         \bool_if:nF
268         {
269             \object_if_local_p:n { #1 } || \object_if_global_p:n { #1 }
270         }
271         {
272             \msg_error:nnx { rawobjects }{ scoperr }{ #1 }
273         }
274     }
275     {
276         \msg_error:nnx { rawobjects }{ noerr }{ #1 }
277     }
278 }
279

```

`\object_member_if_exist_p:nnn`

Tests if the specified member exists

`\object_member_if_exist:nnnTF`

`\object_member_if_exist_p:nn`

`\object_member_if_exist:nnTF`

```

280
281 \prg_new_conditional:Nnn \object_member_if_exist:nnn {p, T, F, TF }
282 {
283     \cs_if_exist:cTF
284     {
285         \object_member_adr:nnn { #1 }{ #2 }{ #3 }
286     }
287     {
288         \prg_return_true:
289     }
290     {
291         \prg_return_false:
292     }
293 }
294
295 \prg_new_conditional:Nnn \object_member_if_exist:nn {p, T, F, TF }
296 {
297     \cs_if_exist:cTF
298     {
299         \object_member_adr:nn { #1 }{ #2 }
300     }
301     {
302         \prg_return_true:
303     }
304     {
305         \prg_return_false:
306     }
307 }
308
309 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nnn
310 { Vnn }{ p, T, F, TF }
311 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nn
312 { Vn }{ p, T, F, TF }

```

313

(End definition for `\object_member_if_exist:nnnTF` and `\object_member_if_exist:nnTF`. These functions are documented on page 8.)

`\object_new_member:nnnNN`  
`\object_new_member:nnn`

Creates a new member variable

```

314
315 \msg_new:nnnn{ rawobjects }{ none }{ Invalid ~ basic ~ type }{ Basic ~ type ~ #1 ~ doesn't
316
317 \cs_new_protected:Nn \object_new_member:nnnNN
318 {
319   \cs_if_exist_use:cTF { #3 _ new:c }
320   {
321     { \object_member_adr:nnnNN { #1 }{ #2 }{ #3 } #4 #5 }
322   }
323   {
324     \msg_error:nnn{ rawobjects }{ none }{ #3 }
325   }
326 }
327
328 \cs_generate_variant:Nn \object_new_member:nnnNN { VnnNN, nnvNN }
329
330 \cs_new_protected:Nn \object_new_member:nnn
331 {
332   \cs_if_exist_use:cTF { #3 _ new:c }
333   {
334     { \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
335   }
336   {
337     \msg_error:nnn{ rawobjects }{ none }{ #3 }
338   }
339 }
340
341 \cs_generate_variant:Nn \object_new_member:nnn { Vnn, nnv }
342

```

(End definition for `\object_new_member:nnnNN` and `\object_new_member:nnn`. These functions are documented on page 8.)

`\object_member_use:nnnNN`  
`\object_member_use:nnn`  
`\object_member_use:nn`

Uses a member variable

```

343
344 \cs_new:Nn \object_member_use:nnnNN
345 {
346   \cs_if_exist_use:cT { #3 _ use:c }
347   {
348     { \object_member_adr:nnnNN { #1 }{ #2 }{ #3 } #4 #5 }
349   }
350 }
351
352 \cs_new:Nn \object_member_use:nnn
353 {
354   \cs_if_exist_use:cT { #3 _ use:c }
355   {
356     { \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
357   }

```



```

358 }
359
360 \cs_new:Nn \object_member_use:nn
361 {
362   \object_member_use:nnv { #1 }{ #2 }
363   {
364     \object_rcmember_adr:nnn { #1 }
365     { #2 _ type }{ str }
366   }
367 }
368
369 \cs_generate_variant:Nn \object_member_use:nnnNN { VnnNN, nnncc }
370 \cs_generate_variant:Nn \object_member_use:nnn { Vnn, vnn, nnv }
371 \cs_generate_variant:Nn \object_member_use:nn { Vn }
372

```

(End definition for `\object_member_use:nnnNN`, `\object_member_use:nnn`, and `\object_member_use:nn`.  
These functions are documented on page 8.)

`\object_member_set:nnnNNn` Set the value a member.

`\object_member_set:nnnn`

`\object_member_set_eq:nnn`

```

373
374 \cs_new_protected:Nn \object_member_set:nnnNNn
375 {
376   \cs_if_exist_use:cT
377   {
378     #3 _ \__rawobjects_scope_pfx:N #4 set:cn
379   }
380   {
381     { \object_member_adr:nnnNN { #1 }{ #2 }{ #3 } #4 #5 }
382     { #6 }
383   }
384 }
385
386 \cs_generate_variant:Nn \object_member_set:nnnNNn { VnnNNn, nnnccn }
387
388 \cs_new_protected:Nn \object_member_set:nnnn
389 {
390   \object_member_set:nnnccn{ #1 }{ #2 }{ #3 }
391   {
392     \object_ncmember_adr:nnn
393     {
394       \object_embedded_adr:nn{ #1 }{ /_I_/ }
395     }
396     { S }{ str }
397   }
398   {
399     \object_ncmember_adr:nnn
400     {
401       \object_embedded_adr:nn{ #1 }{ /_I_/ }
402     }
403     { V }{ str }
404   }
405   { #4 }
406 }

```

```

407
408 \cs_generate_variant:Nn \object_member_set:nnnn { Vnnn, nnvn }
409
410 \cs_new_protected:Nn \object_member_set:nnn
411 {
412   \object_member_set:nnvn { #1 } { #2 }
413   {
414     \object_rcmember_adr:nnn { #1 }
415     { #2 _ type } { str }
416   } { #3 }
417 }
418
419 \cs_generate_variant:Nn \object_member_set:nnn { Vnn }
420

```

(End definition for `\object_member_set:nnnNn`, `\object_member_set:nnnn`, and `\object_member_set_eq:nnn`. These functions are documented on page 9.)

`\object_member_set_eq:nnnN`  
`\object_member_set_eq:nnN`

Make a member equal to another variable.

```

421
422 \cs_new_protected:Nn \object_member_set_eq:nnnN
423 {
424   \__rawobjects_force_scope:n { #1 }
425   \cs_if_exist_use:cT
426   {
427     #3 _ \__rawobjects_scope_pfx:n { #1 } set _ eq:cN
428   }
429   {
430     { \object_member_adr:nnn { #1 } { #2 } { #3 } } #4
431   }
432 }
433
434 \cs_generate_variant:Nn \object_member_set_eq:nnnN { VnnN, nnnnc, Vnnnc, nnvN }
435
436 \cs_new_protected:Nn \object_member_set_eq:nnN
437 {
438   \object_member_set_eq:nnvN { #1 } { #2 }
439   {
440     \object_rcmember_adr:nnn { #1 }
441     { #2 _ type } { str }
442   } #3
443 }
444
445 \cs_generate_variant:Nn \object_member_set_eq:nnN { VnN, nnc, Vnc }
446

```

(End definition for `\object_member_set_eq:nnnN` and `\object_member_set_eq:nnN`. These functions are documented on page 9.)

`\object_ncmember_adr:nnn`

Get address of near constant

```

447
448 \cs_new:Nn \object_ncmember_adr:nnn
449 {
450   \tl_to_str:n { c _ } #1 \tl_to_str:n { _ CONST _ #2 _ #3 }
451 }

```

```

452
453 \cs_generate_variant:Nn \object_ncmember_adr:nnn { Vnn, vnn }
454

```

(End definition for \object\_ncmember\_adr:nnn. This function is documented on page 9.)

**\object\_rcmember\_adr:nnn** Get the address of a remote constant.

```

455
456 \cs_new:Nn \object_rcmember_adr:nnn
457 {
458   \object_ncmember_adr:vnn
459   {
460     \object_ncmember_adr:nnn
461     {
462       \object_embedded_adr:nn{ #1 }{ /_I_/ }
463     }
464     { P }{ str }
465   }
466   { #2 }{ #3 }
467 }
468
469 \cs_generate_variant:Nn \object_rcmember_adr:nnn { Vnn }

```

(End definition for \object\_rcmember\_adr:nnn. This function is documented on page 9.)

**\object\_ncmember\_if\_exist:p:nnn** Tests if the specified member constant exists.

**\object\_ncmember\_if\_exist:nnnTF**  
**\object\_rcmember\_if\_exist:p:nnn**  
**\object\_rcmember\_if\_exist:nnnTF**

```

470
471 \prg_new_conditional:Nnn \object_ncmember_if_exist:nnn {p, T, F, TF }
472 {
473   \cs_if_exist:cTF
474   {
475     \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 }
476   }
477   {
478     \prg_return_true:
479   }
480   {
481     \prg_return_false:
482   }
483 }
484
485 \prg_new_conditional:Nnn \object_rcmember_if_exist:nnn {p, T, F, TF }
486 {
487   \cs_if_exist:cTF
488   {
489     \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 }
490   }
491   {
492     \prg_return_true:
493   }
494   {
495     \prg_return_false:
496   }
497 }
498

```

```

499 \prg_generate_conditional_variant:Nnn \object_ncmember_if_exist:nnn
500   { Vnn }{ p, T, F, TF }
501 \prg_generate_conditional_variant:Nnn \object_rcmember_if_exist:nnn
502   { Vnn }{ p, T, F, TF }
503

```

(End definition for `\object_ncmember_if_exist:nnnTF` and `\object_rcmember_if_exist:nnnTF`. These functions are documented on page 9.)

`\object_ncmember_use:nnn` Uses a near/remote constant.

`\object_rcmember_use:nnn`

```

504
505 \cs_new:Nn \object_ncmember_use:nnn
506   {
507     \cs_if_exist_use:cT { #3 _ use:c }
508     {
509       { \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 } }
510     }
511   }
512
513 \cs_new:Nn \object_rcmember_use:nnn
514   {
515     \cs_if_exist_use:cT { #3 _ use:c }
516     {
517       { \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 } }
518     }
519   }
520
521 \cs_generate_variant:Nn \object_ncmember_use:nnn { Vnn }
522 \cs_generate_variant:Nn \object_rcmember_use:nnn { Vnn }
523

```

(End definition for `\object_ncmember_use:nnn` and `\object_rcmember_use:nnn`. These functions are documented on page 9.)

`\object_newconst:nnnn` Creates a constant variable, use with caution

```

524
525 \cs_new_protected:Nn \object_newconst:nnnn
526   {
527     \use:c { #3 _ const:cn }
528     {
529       \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 }
530     }
531     { #4 }
532   }
533

```

(End definition for `\object_newconst:nnnn`. This function is documented on page 11.)

`\object_newconst_tl:nnn` Create constants

`\object_newconst_str:nnn`

`\object_newconst_int:nnn`

`\object_newconst_clist:nnn`

`\object_newconst_dim:nnn`

`\object_newconst_skip:nnn`

`\object_newconst_fp:nnn`

```

534
535 \cs_new_protected:Nn \object_newconst_tl:nnn
536   {
537     \object_newconst:nnnn { #1 }{ #2 }{ tl }{ #3 }
538   }
539 \cs_new_protected:Nn \object_newconst_str:nnn

```

```

540 {
541   \object_newconst:nnnn { #1 }{ #2 }{ str }{ #3 }
542 }
543 \cs_new_protected:Nn \object_newconst_int:nnn
544 {
545   \object_newconst:nnnn { #1 }{ #2 }{ int }{ #3 }
546 }
547 \cs_new_protected:Nn \object_newconst_clist:nnn
548 {
549   \object_newconst:nnnn { #1 }{ #2 }{ clist }{ #3 }
550 }
551 \cs_new_protected:Nn \object_newconst_dim:nnn
552 {
553   \object_newconst:nnnn { #1 }{ #2 }{ dim }{ #3 }
554 }
555 \cs_new_protected:Nn \object_newconst_skip:nnn
556 {
557   \object_newconst:nnnn { #1 }{ #2 }{ skip }{ #3 }
558 }
559 \cs_new_protected:Nn \object_newconst_fp:nnn
560 {
561   \object_newconst:nnnn { #1 }{ #2 }{ fp }{ #3 }
562 }
563
564 \cs_generate_variant:Nn \object_newconst_tl:nnn { Vnn }
565 \cs_generate_variant:Nn \object_newconst_str:nnn { Vnn }
566 \cs_generate_variant:Nn \object_newconst_int:nnn { Vnn }
567 \cs_generate_variant:Nn \object_newconst_clist:nnn { Vnn }
568 \cs_generate_variant:Nn \object_newconst_dim:nnn { Vnn }
569 \cs_generate_variant:Nn \object_newconst_skip:nnn { Vnn }
570 \cs_generate_variant:Nn \object_newconst_fp:nnn { Vnn }
571
572
573 \cs_generate_variant:Nn \object_newconst_str:nnn { nnx }
574 \cs_generate_variant:Nn \object_newconst_str:nnn { nnV }
575

```

(End definition for `\object_newconst_tl:nnn` and others. These functions are documented on page 10.)

`\object_newconst_seq_from_clist:nnn` Creates a seq constant.

```

576
577 \cs_new_protected:Nn \object_newconst_seq_from_clist:nnn
578 {
579   \seq_const_from_clist:cn
580   {
581     \object_ncmember_adr:nnn { #1 }{ #2 }{ seq }
582   }
583   { #3 }
584 }
585
586 \cs_generate_variant:Nn \object_newconst_seq_from_clist:nnn { Vnn }
587

```

(End definition for `\object_newconst_seq_from_clist:nnn`. This function is documented on page 11.)

`\object_newconst_prop_from_keyval:nnn` Creates a prop constant.

```

588
589 \cs_new_protected:Nn \object_newconst_prop_from_keyval:nnn
590 {
591   \prop_const_from_keyval:cn
592   {
593     \object_ncmember_adr:nnn { #1 }{ #2 }{ prop }
594   }
595   { #3 }
596 }
597
598 \cs_generate_variant:Nn \object_newconst_prop_from_keyval:nnn { Vnn }
599

```

(End definition for `\object_newconst_prop_from_keyval:nnn`. This function is documented on page 11.)

`\object_ncmethod_adr:nnn` Fully expands to the method address.

`\object_rcmethod_adr:nnn`

```

600
601 \cs_new:Nn \object_ncmethod_adr:nnn
602 {
603   #1 \tl_to_str:n { _ CMETHOD _ #2 : #3 }
604 }
605
606 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
607
608 \cs_new:Nn \object_rcmethod_adr:nnn
609 {
610   \object_ncmethod_adr:vnn
611   {
612     \object_ncmember_adr:nnn
613     {
614       \object_embedded_adr:nn{ #1 }{ /_I_/ }
615     }
616     { P }{ str }
617   }
618   { #2 }{ #3 }
619 }
620
621 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
622 \cs_generate_variant:Nn \object_rcmethod_adr:nnn { Vnn }
623

```

(End definition for `\object_ncmethod_adr:nnn` and `\object_rcmethod_adr:nnn`. These functions are documented on page 9.)

`\object_ncmethod_if_exist_p:nnn` Tests if the specified member constant exists.

`\object_ncmethod_if_exist:nnnTF`

`\object_rcmethod_if_exist_p:nnn`

`\object_rcmethod_if_exist:nnnTF`

```

624
625 \prg_new_conditional:Nnn \object_ncmethod_if_exist:nnn {p, T, F, TF }
626 {
627   \cs_if_exist:cTF
628   {
629     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
630   }
631   {

```

```

632         \prg_return_true:
633     }
634     {
635         \prg_return_false:
636     }
637 }
638
639 \prg_new_conditional:Nnn \object_rcmethod_if_exist:nnn {p, T, F, TF }
640 {
641     \cs_if_exist:cTF
642     {
643         \object_rcmethodr_adr:nnn { #1 }{ #2 }{ #3 }
644     }
645     {
646         \prg_return_true:
647     }
648     {
649         \prg_return_false:
650     }
651 }
652
653 \prg_generate_conditional_variant:Nnn \object_ncmethod_if_exist:nnn
654 { Vnn }{ p, T, F, TF }
655 \prg_generate_conditional_variant:Nnn \object_rcmethod_if_exist:nnn
656 { Vnn }{ p, T, F, TF }
657

```

(End definition for `\object_ncmethod_if_exist:nnnTF` and `\object_rcmethod_if_exist:nnnTF`. These functions are documented on page 10.)

**`\object_new_cmethod:nnnn`** Creates a new method

```

658
659 \cs_new_protected:Nn \object_new_cmethod:nnnn
660 {
661     \cs_new:cn
662     {
663         \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
664     }
665     { #4 }
666 }
667
668 \cs_generate_variant:Nn \object_new_cmethod:nnnn { Vnnn }
669

```

(End definition for `\object_new_cmethod:nnnn`. This function is documented on page 10.)

**`\object_ncmethod_call:nnn`** Calls the specified method.

**`\object_rcmethod_call:nnn`**

```

670
671 \cs_new:Nn \object_ncmethod_call:nnn
672 {
673     \use:c
674     {
675         \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
676     }
677 }

```

```

678
679 \cs_new:Nn \object_rcmethod_call:nnn
680 {
681   \use:c
682   {
683     \object_rcmethod_adr:nnn { #1 }{ #2 }{ #3 }
684   }
685 }
686
687 \cs_generate_variant:Nn \object_ncmethod_call:nnn { Vnn }
688 \cs_generate_variant:Nn \object_rcmethod_call:nnn { Vnn }
689

```

(End definition for `\object_ncmethod_call:nnn` and `\object_rcmethod_call:nnn`. These functions are documented on page 10.)

**\proxy\_init\_SV:nnnNN** Initialize scope and visibility variables

```

690
691 \cs_new_protected:Nn \proxy_init_SV:nnnNN
692 {
693   \object_newconst_str:nnV
694   {
695     \object_embedded_adr:nn{ #3 }{ /_I_/ }
696   }
697   { S } #4
698   \object_newconst_str:nnV
699   {
700     \object_embedded_adr:nn{ #3 }{ /_I_/ }
701   }
702   { V } #5
703 }
704
705 \cs_generate_variant:Nn \proxy_init_SV:nnnNN { nnVNN, nVVNN }
706

```

(End definition for `\proxy_init_SV:nnnNN`. This function is documented on page 14.)

**\proxy\_init\_MP:nnnNN** Initialize module name and proxy name variables

```

707
708 \cs_new_protected:Nn \proxy_init_MP:nnnNN
709 {
710   \object_newconst_str:nnn
711   {
712     \object_embedded_adr:nn{ #3 }{ /_I_/ }
713   }
714   { M }{ #2 }
715   \object_newconst_str:nnn
716   {
717     \object_embedded_adr:nn{ #3 }{ /_I_/ }
718   }
719   { P }{ #1 }
720 }
721
722 \cs_generate_variant:Nn \proxy_init_MP:nnnNN { nnVNN, nVVNN }
723

```



(End definition for \proxy\_init\_MP:nnnNN. This function is documented on page 14.)

```

724
725 \cs_new_protected:Nn \__rawobjects_initproxy:nnnNN
726 {
727   \object_newconst_str:nnn
728   {
729     \object_embedded_adr:nn{ #3 }{ /_I_/ }
730   }
731   { M }{ #2 }
732   \object_newconst_str:nnn
733   {
734     \object_embedded_adr:nn{ #3 }{ /_I_/ }
735   }
736   { P }{ #1 }
737   \object_newconst_str:nnV
738   {
739     \object_embedded_adr:nn{ #3 }{ /_I_/ }
740   }
741   { S } #4
742   \object_newconst_str:nnV
743   {
744     \object_embedded_adr:nn{ #3 }{ /_I_/ }
745   }
746   { V } #5
747   \object_newconst:nnnn
748   {
749     \object_embedded_adr:nn{ #3 }{ /_I_/ }
750   }
751   { ifprox }{ bool }{ \c_true_bool }
752 }
753 \cs_generate_variant:Nn \__rawobjects_initproxy:nnnNN { VnVNN }
754
755 \cs_new_protected:Nn \__rawobjects_initproxy_lg:nnnnNN
756 {
757   \proxy_add_initializer:nN { #3 }
758   \proxy_init_SV:nnnNN
759 }
760
761 \cs_new_protected:Nn \__rawobjects_initproxy_all:nnnnNN
762 {
763   \proxy_add_initializer:nN { #3 }
764   \proxy_init_SV:nnnNN
765   \proxy_add_initializer:nN { #3 }
766   \proxy_init_MP:nnnNN
767 }
768

```

\object\_if\_proxy\_p:n Test if an object is a proxy.  
 \object\_if\_proxy:nTF

```

769
770 \cs_new:Nn \__rawobjects_bol_com:N
771 {
772   \cs_if_exist_p:N #1 && \bool_if_p:N #1
773 }
774

```

```

775 \cs_generate_variant:Nn \_rawobjects_bol_com:N { c }
776
777 \prg_new_conditional:Nnn \object_if_proxy:n {p, T, F, TF}
778 {
779   \cs_if_exist:cTF
780   {
781     \object_ncmember_adr:nnn
782     {
783       \object_embedded_adr:nn{ #1 }{ /_I_/ }
784     }
785     { ifprox }{ bool }
786   }
787   {
788     \bool_if:cTF
789     {
790       \object_ncmember_adr:nnn
791       {
792         \object_embedded_adr:nn{ #1 }{ /_I_/ }
793       }
794       { ifprox }{ bool }
795     }
796     {
797       \prg_return_true:
798     }
799     {
800       \prg_return_false:
801     }
802   }
803   {
804     \prg_return_false:
805   }
806 }
807

```

(End definition for \object\_if\_proxy:nTF. This function is documented on page 11.)

```

\object_test_proxy_p:nn Test if an object is generated from selected proxy.
\object_test_proxy:nnTF
\object_test_proxy_p:nN
\object_test_proxy:nNTF
808
809 \prg_generate_conditional_variant:Nnn \str_if_eq:nn { ve }{ TF }
810
811 \prg_new_conditional:Nnn \object_test_proxy:nn {p, T, F, TF}
812 {
813   \str_if_eq:veTF
814   {
815     \object_ncmember_adr:nnn
816     {
817       \object_embedded_adr:nn{ #1 }{ /_I_/ }
818     }
819     { P }{ str }
820   }
821   { #2 }
822   {
823     \prg_return_true:
824   }

```

```

825     {
826       \prg_return_false:
827     }
828   }
829
830 \prg_new_conditional:Nnn \object_test_proxy:nN {p, T, F, TF}
831 {
832   \str_if_eq:cNTF
833   {
834     \object_ncmember_adr:nnn
835     {
836       \object_embedded_adr:nn{ #1 }{ /_I_/ }
837     }
838     { P }{ str }
839   }
840   #2
841   {
842     \prg_return_true:
843   }
844   {
845     \prg_return_false:
846   }
847 }
848
849 \prg_generate_conditional_variant:Nnn \object_test_proxy:nn
850 { Vn }{p, T, F, TF}
851 \prg_generate_conditional_variant:Nnn \object_test_proxy:nN
852 { VN }{p, T, F, TF}
853

```

(End definition for `\object_test_proxy:nnTF` and `\object_test_proxy:nNTF`. These functions are documented on page [12](#).)

```

\object_create:nnnNN Creates an object from a proxy.
\object_create_set:NnnnNN
\object_create_gset:NnnnNN
\embedded_create:nnnnNN
\embedded_create:nnn

854
855 \msg_new:nnnn { rawobjects }{ notproxy }{ Fake ~ proxy }
856 {
857   Object ~ #1 ~ is ~ not ~ a ~ proxy.
858 }
859
860 \cs_new_protected:Nn \__rawobjects_force_proxy:n
861 {
862   \object_if_proxy:nF { #1 }
863   {
864     \msg_error:nnn { rawobjects }{ notproxy }{ #1 }
865   }
866 }
867
868 \cs_new_protected:Nn \__rawobjects_create_anon:nnnNN
869 {
870   \tl_if_empty:nF{ #1 }
871   {
872
873     \__rawobjects_force_proxy:n { #1 }

```

```

874
875 \seq_map_inline:cn
876 {
877   \object_member_adr:nnn { #1 }{ varlist }{ seq }
878 }
879 {
880   \object_new_member:nnvNN { #3 }{ ##1 }
881   {
882     \object_ncmember_adr:nnn { #1 }{ ##1 _ type }{ str }
883   }
884   #4 #5
885 }
886
887 \seq_map_inline:cn
888 {
889   \object_member_adr:nnn { #1 }{ objlist }{ seq }
890 }
891 {
892   \embedded_create:nvnnNN
893   { #3 }
894   {
895     \object_ncmember_adr:nnn { #1 }{ ##1 _ proxy }{ str }
896   }
897   { #2 }{ ##1 } #4 #5
898 }
899
900 \tl_map_inline:cn
901 {
902   \object_member_adr:nnn { #1 }{ init }{ tl }
903 }
904 {
905   ##1 { #1 }{ #2 }{ #3 } #4 #5
906 }
907
908 }
909 }
910
911 \cs_generate_variant:Nn \__rawobjects_create_anon:nnnNN { xnxNN, VnVNN }
912
913 \cs_new_protected:Nn \object_create:nnnNN
914 {
915   \__rawobjects_create_anon:xnxNN { #1 }{ #2 }
916   { \object_address:nn { #2 }{ #3 } }
917   #4 #5
918 }
919
920 \cs_generate_variant:Nn \object_create:nnnNN { VnnNN }
921
922 \cs_new_protected:Nn \object_create_set:NnnnNN
923 {
924   \object_create:nnnNN { #2 }{ #3 }{ #4 } #5 #6
925   \str_set:Nx #1 { \object_address:nn { #3 }{ #4 } }
926 }
927

```

```

928 \cs_new_protected:Nn \object_create_gset:NnnnNN
929 {
930   \object_create:nnnNN { #2 } { #3 } { #4 } #5 #6
931   \str_gset:Nx #1 { \object_address:nn { #3 } { #4 } }
932 }
933
934 \cs_generate_variant:Nn \object_create_set:NnnnNN { NVnnNN, NnnfNN }
935 \cs_generate_variant:Nn \object_create_gset:NnnnNN { NVnnNN, NnnfNN }
936
937 \cs_new_protected:Nn \embedded_create:nnnnNN
938 {
939   \__rawobjects_create_anon:xnxNN { #2 }
940   { #3 }
941   {
942     \object_embedded_adr:nn
943     { #1 } { #4 }
944   }
945   #5 #6
946 }
947
948 \cs_generate_variant:Nn \embedded_create:nnnnNN { nvnnNN, nnvncc }
949
950 \cs_new_protected:Nn \embedded_create:nnn
951 {
952   \embedded_create:nnvncc { #1 } { #2 }
953   {
954     \object_ncmember_adr:nnn
955     {
956       \object_embedded_adr:nn{ #1 } { /_I_/ }
957     }
958     { M } { str }
959   }
960   { #3 }
961   {
962     \object_ncmember_adr:nnn
963     {
964       \object_embedded_adr:nn{ #1 } { /_I_/ }
965     }
966     { S } { str }
967   }
968   {
969     \object_ncmember_adr:nnn
970     {
971       \object_embedded_adr:nn{ #1 } { /_I_/ }
972     }
973     { V } { str }
974   }
975 }
976
977 \cs_generate_variant:Nn \embedded_create:nnn { nvnn, Vnn }
978

```

(End definition for `\object_create:nnnNN` and others. These functions are documented on page 12.)

`\proxy_create:nnN`  
`\proxy_create_set:NnnN`  
`\proxy_create_gset:NnnN`

Creates a new proxy object

```

979
980 \cs_new_protected:Nn \proxy_create:nnN
981 {
982   \object_create:VnnNN \c_proxy_address_str { #1 }{ #2 }
983   \c_object_global_str #3
984 }
985
986 \cs_new_protected:Nn \proxy_create_set:NnnN
987 {
988   \object_create_set:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
989   \c_object_global_str #4
990 }
991
992 \cs_new_protected:Nn \proxy_create_gset:NnnN
993 {
994   \object_create_gset:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
995   \c_object_global_str #4
996 }
997

```

(End definition for `\proxy_create:nnN`, `\proxy_create_set:NnnN`, and `\proxy_create_gset:NnnN`. These functions are documented on page 13.)

`\proxy_push_member:nnn`

Push a new member inside a proxy.

```

998
999 \cs_new_protected:Nn \proxy_push_member:nnn
1000 {
1001   \object_newconst_str:nnn { #1 }{ #2 _ type }{ #3 }
1002   \seq_gput_left:cn
1003   {
1004     \object_member_adr:nnn { #1 }{ varlist }{ seq }
1005   }
1006   { #2 }
1007 }
1008
1009 \cs_generate_variant:Nn \proxy_push_member:nnn { Vnn }
1010

```

(End definition for `\proxy_push_member:nnn`. This function is documented on page 13.)

`\proxy_push_embedded:nnn`

Push a new embedded object inside a proxy.

```

1011
1012 \cs_new_protected:Nn \proxy_push_embedded:nnn
1013 {
1014   \object_newconst_str:nnx { #1 }{ #2 _ proxy }{ #3 }
1015   \seq_gput_left:cn
1016   {
1017     \object_member_adr:nnn { #1 }{ objlist }{ seq }
1018   }
1019   { #2 }
1020 }
1021
1022 \cs_generate_variant:Nn \proxy_push_embedded:nnn { Vnn }
1023

```

(End definition for `\proxy_push_embedded:nnn`. This function is documented on page 13.)

`\proxy_add_initializer:nN` Push a new embedded object inside a proxy.

```

1024
1025 \cs_new_protected:Nn \proxy_add_initializer:nN
1026 {
1027   \tl_gput_right:cn
1028   {
1029     \object_member_adr:nnn { #1 }{ init }{ t1 }
1030   }
1031   { #2 }
1032 }
1033
1034 \cs_generate_variant:Nn \proxy_add_initializer:nN { VN }
1035

```

(End definition for `\proxy_add_initializer:nN`. This function is documented on page 14.)

`\c_proxy_min_adr_str` Variable containing the address of the proxymin object.

```

1036 \str_const:Nx \c_proxy_min_adr_str
1037 { \object_address:nn { rawobjects }{ proxymin } }
1038
1039
1040 \__rawobjects_initproxy:VnVNN \c_proxy_min_adr_str { rawobjects } \c_proxy_min_adr_str
1041 \c_object_global_str \c_object_public_str
1042
1043 \object_new_member:VnnNN \c_proxy_min_adr_str { init }{ t1 }
1044 \c_object_global_str \c_object_public_str
1045
1046 \object_new_member:VnnNN \c_proxy_min_adr_str { varlist }{ seq }
1047 \c_object_global_str \c_object_public_str
1048
1049 \object_new_member:VnnNN \c_proxy_min_adr_str { objlist }{ seq }
1050 \c_object_global_str \c_object_public_str
1051
1052 \proxy_push_member:Vnn \c_proxy_min_adr_str
1053 { init }{ t1 }
1054 \proxy_push_member:Vnn \c_proxy_min_adr_str
1055 { varlist }{ seq }
1056 \proxy_push_member:Vnn \c_proxy_min_adr_str
1057 { objlist }{ seq }
1058
1059 \proxy_add_initializer:VN \c_proxy_min_adr_str
1060 \__rawobjects_initproxy:nnnNN
1061

```

(End definition for `\c_proxy_min_adr_str`. This variable is documented on page 12.)

`\c_proxy_empty_adr_str` A proxy object that initialize only the needed internal variables, useful to create empty objects.

```

1062 \str_const:Nx \c_proxy_empty_adr_str
1063 { \object_address:nn { rawobjects }{ empty } }
1064
1065 \__rawobjects_create_anon:VnVNN \c_proxy_min_adr_str { rawobjects } \c_proxy_empty_adr_str

```

```

1066 \c_object_global_str \c_object_public_str
1067
1068 \proxy_add_initializer:VN \c_proxy_empty_adr_str
1069 \proxy_init_SV:nnnNN
1070 \proxy_add_initializer:VN \c_proxy_empty_adr_str
1071 \proxy_init_MP:nnnNN
1072

```

(End definition for \c\_proxy\_empty\_adr\_str. This variable is documented on page 12.)

**\c\_proxy\_lgt\_adr\_str** Variable containing the address of the proxylgt object.

```

1073 \str_const:Nx \c_proxy_lgt_adr_str
1074 { \object_address:nn { rawobjects }{ proxylgt } }
1075
1076
1077 \__rawobjects_create_anon:VnVNN \c_proxy_min_adr_str { rawobjects } \c_proxy_lgt_adr_str
1078 \c_object_global_str \c_object_public_str
1079
1080 \proxy_push_member:Vnn \c_proxy_lgt_adr_str
1081 { init }{ tl }
1082 \proxy_push_member:Vnn \c_proxy_lgt_adr_str
1083 { varlist }{ seq }
1084 \proxy_push_member:Vnn \c_proxy_lgt_adr_str
1085 { objlist }{ seq }
1086
1087 \proxy_add_initializer:VN \c_proxy_lgt_adr_str
1088 \__rawobjects_initproxy:nnnNN
1089 \proxy_add_initializer:VN \c_proxy_lgt_adr_str
1090 \__rawobjects_initproxy_lg:nnnNN
1091

```

(End definition for \c\_proxy\_lgt\_adr\_str. This variable is documented on page 12.)

**\c\_proxy\_address\_str** Variable containing the address of the proxy object.

```

1092 \str_const:Nx \c_proxy_address_str
1093 { \object_address:nn { rawobjects }{ proxy } }
1094
1095
1096 \__rawobjects_create_anon:VnVNN \c_proxy_min_adr_str { rawobjects } \c_proxy_address_str
1097 \c_object_global_str \c_object_public_str
1098
1099 \proxy_push_member:Vnn \c_proxy_address_str
1100 { init }{ tl }
1101 \proxy_push_member:Vnn \c_proxy_address_str
1102 { varlist }{ seq }
1103 \proxy_push_member:Vnn \c_proxy_address_str
1104 { objlist }{ seq }
1105
1106 \proxy_add_initializer:VN \c_proxy_address_str
1107 \__rawobjects_initproxy:nnnNN
1108 \proxy_add_initializer:VN \c_proxy_address_str
1109 \__rawobjects_initproxy_all:nnnNN
1110

```

(End definition for \c\_proxy\_address\_str. This variable is documented on page 12.)



**\object\_allocate\_incr:NNnnNN**

**\object\_gallocate\_incr:NNnnNN**

**\object\_allocate\_gincr:NNnnNN**

**\object\_gallocate\_gincr:NNnnNN**

Create an address and use it to instantiate an object

```
1111
1112 \cs_new:Nn \__rawobjects_combine_aux:nnn
1113 {
1114     anon . #3 . #2 . #1
1115 }
1116
1117 \cs_generate_variant:Nn \__rawobjects_combine_aux:nnn { Vnf }
1118
1119 \cs_new:Nn \__rawobjects_combine:Nn
1120 {
1121     \__rawobjects_combine_aux:Vnf #1 { #2 }
1122     {
1123         \cs_to_str:N #1
1124     }
1125 }
1126
1127 \cs_new_protected:Nn \object_allocate_incr:NNnnNN
1128 {
1129     \object_create_set:NnnfNN #1 { #3 }{ #4 }
1130     {
1131         \__rawobjects_combine:Nn #2 { #3 }
1132     }
1133     #5 #6
1134
1135     \int_incr:N #2
1136 }
1137
1138 \cs_new_protected:Nn \object_gallocate_incr:NNnnNN
1139 {
1140     \object_create_gset:NnnfNN #1 { #3 }{ #4 }
1141     {
1142         \__rawobjects_combine:Nn #2 { #3 }
1143     }
1144     #5 #6
1145
1146     \int_incr:N #2
1147 }
1148
1149 \cs_generate_variant:Nn \object_allocate_incr:NNnnNN { NNVnNN }
1150
1151 \cs_generate_variant:Nn \object_gallocate_incr:NNnnNN { NNVnNN }
1152
1153 \cs_new_protected:Nn \object_allocate_gincr:NNnnNN
1154 {
1155     \object_create_set:NnnfNN #1 { #3 }{ #4 }
1156     {
1157         \__rawobjects_combine:Nn #2 { #3 }
1158     }
1159     #5 #6
1160
1161     \int_gincr:N #2
1162 }
1163
```

```

1164 \cs_new_protected:Nn \object_gallocate_gincr:NNnnNN
1165 {
1166   \object_create_gset:NnnfNN #1 { #3 }{ #4 }
1167   {
1168     \__rawobjects_combine:Nn #2 { #3 }
1169   }
1170   #5 #6
1171
1172   \int_gincr:N #2
1173 }
1174
1175 \cs_generate_variant:Nn \object_allocate_gincr:NNnnNN { NNvNNN }
1176
1177 \cs_generate_variant:Nn \object_gallocate_gincr:NNnnNN { NNvNNN }
1178

```

(End definition for `\object_allocate_incr:NNnnNN` and others. These functions are documented on page 13.)

**`\object_assign:nn`** Copy an object to another one.

```

1179 \cs_new_protected:Nn \object_assign:nn
1180 {
1181   \seq_map_inline:cn
1182   {
1183     \object_member_adr:vnn
1184     {
1185       \object_ncmember_adr:nnn
1186       {
1187         \object_embedded_adr:nn{ #1 }{ /_I_/ }
1188       }
1189       { P }{ str }
1190     }
1191     { varlist }{ seq }
1192   }
1193   {
1194     \object_member_set_eq:nnc { #1 }{ ##1 }
1195     {
1196       \object_member_adr:nn{ #2 }{ ##1 }
1197     }
1198   }
1199 }
1200
1201 \cs_generate_variant:Nn \object_assign:nn { nV, Vn, VV }

```

(End definition for `\object_assign:nn`. This function is documented on page 14.)

```

1202 </package>

```