

The lt3rawobjects package

Paolo De Donato

Released on 2022/12/09 Version 2.2

Contents

1	Introduction	1
2	Objects and proxies	2
3	Put objects inside objects	3
3.1	Put a pointer variable	3
3.2	Clone the inner structure	4
3.3	Embedded objects	5
4	Library functions	5
4.1	Base object functions	5
4.2	Members	6
4.3	Methods	8
4.4	Constant member creation	9
4.5	Macros	10
4.6	Proxy utilities and object creation	10
5	Examples	12
6	Templated proxies	14
7	Implementation	15

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 Instead changes introduced in minor and patch version updates are always backward compatible, and any withdrawn function is declared deprecated instead of being removed.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^AT_EX3 guidelines about variables management, for additional motivations you can see [this thread](#) in the L^AT_EX3 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 `tl` or `str`) holding the newly created address:

```
\object_new_member:nnn
{
  \object_address:nn{ mymod }{ A }
}{ pointer }{ tl }

\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 emmbedded 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 Library functions

4.1 Base object functions

<code>\object_address:nn</code> ★	<code>\object_address:nn {<module>} {<id>}</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_gset:Nnn</code>	<code>\object_address_set:nn <str var> {<module>} {<id>}</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 {⟨address⟩} {⟨id⟩}</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

<code>\object_if_exist_p:n</code>	★	<code>\object_if_exist_p:n {⟨address⟩}</code>
<code>\object_if_exist_p:V</code>	★	<code>\object_if_exist:nTF {⟨address⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_if_exist:nTF</code>	★	Tests if an object was instantiated at the specified address.
<code>\object_if_exist:VTF</code>	★	From: 1.0

<code>\object_get_module:n</code>	★	<code>\object_get_module:n {⟨address⟩}</code>
<code>\object_get_module:V</code>	★	<code>\object_get_proxy_adr:n {⟨address⟩}</code>
<code>\object_get_proxy_adr:n</code>	★	Get the object module and its generator.
<code>\object_get_proxy_adr:V</code>	★	From: 1.0

<code>\object_if_local_p:n</code>	★	<code>\object_if_local_p:n {⟨address⟩}</code>
<code>\object_if_local_p:V</code>	★	<code>\object_if_local:nTF {⟨address⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_if_local:nTF</code>	★	Tests if the object is local or global.
<code>\object_if_local:VTF</code>	★	From: 1.0
<code>\object_if_global_p:n</code>	★	
<code>\object_if_global_p:V</code>	★	
<code>\object_if_global:nTF</code>	★	
<code>\object_if_global:VTF</code>	★	

<code>\object_if_public_p:n</code>	★	<code>\object_if_public_p:n {⟨address⟩}</code>
<code>\object_if_public_p:V</code>	★	<code>\object_if_public:nTF {⟨address⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_if_public:nTF</code>	★	Tests if the object is public or private.
<code>\object_if_public:VTF</code>	★	From: 1.0
<code>\object_if_private_p:n</code>	★	
<code>\object_if_private_p:V</code>	★	
<code>\object_if_private:nTF</code>	★	
<code>\object_if_private:VTF</code>	★	

4.2 Members

<code>\object_member_adr:nnn</code>	★	<code>\object_member_adr:nnn {⟨address⟩} {⟨member name⟩} {⟨member type⟩}</code>
<code>\object_member_adr:(Vnn nnv)</code>	★	<code>\object_member_adr:nn {⟨address⟩} {⟨member name⟩}</code>
<code>\object_member_adr:nn</code>	★	
<code>\object_member_adr:Vn</code>	★	

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

<code>\object_member_if_exist_p:nnn</code>	<code>*</code>	<code>\object_member_if_exist_p:nnn {<address>} {<member name>} {<member type>}</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 {<address>} {<member name>} {<member type>} {<true code>} {<false code>}</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 {<address>} {<member name>}</code>
<code>\object_member_if_exist_p:Vn</code>	<code>*</code>	<code>\object_member_if_exist_p:nnTF {<address>} {<member name>} {<true code>}</code>
<code>\object_member_if_exist:nnTF</code>	<code>*</code>	<code>{<false code>}</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 {<address>} {<member name>}</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 {<address>} {<member name>} {<member type>}</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_member_use:nnn</code>	<code>*</code>	<code>\object_member_use:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_member_use:(Vnn nnv)</code>	<code>*</code>	<code>\object_member_use:nn {<address>} {<member name>}</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_set:nnnn</code>	<code>\object_member_set:nnnn {<address>} {<member name>} {<member type>}</code>
<code>\object_member_set:(nnvn Vnnn)</code>	<code>{<value>}</code>
<code>\object_member_set:nnn</code>	<code>\object_member_set:nnn {<address>} {<member name>} {<value>}</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_eq:nnnN</code>	<code>\object_member_set_eq:nnnN {<address>} {<member name>}</code>
<code>\object_member_set_eq:(nnvN VnnN nnnc Vnnc)</code>	<code>{<member type>} <variable></code>
<code>\object_member_set_eq:nnN</code>	<code>\object_member_set_eq:nnN {<address>} {<member name>}</code>
<code>\object_member_set_eq:(VnN nnc Vnc)</code>	<code><variable></code>

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

From: 1.0

<code>\object_ncmper_adr:nnn</code>	<code>*</code>	<code>\object_ncmper_adr:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_ncmper_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_ncmper_if_exist_p:nnn</code>	<code>*</code>	<code>\object_ncmper_if_exist_p:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_ncmper_if_exist_p:Vnn</code>	<code>*</code>	
<code>\object_ncmper_if_exist:nnnTF</code>	<code>*</code>	<code>\object_ncmper_if_exist:nnnTF {<address>} {<member name>} {<member type>} {<true code>} {<false code>}</code>
<code>\object_ncmper_if_exist:VnnTF</code>	<code>*</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_ncmper_use:nnn</code>	<code>*</code>	<code>\object_ncmper_use:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_ncmper_use:Vnn</code>	<code>*</code>	
<code>\object_rcmember_use:nnn</code>	<code>*</code>	
<code>\object_rcmember_use:Vnn</code>	<code>*</code>	

Uses the specified near/remote constant member.

From: 2.0

4.3 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 {<address>} {<method name>} {<method variant>}</code>
<code>\object_ncmethod_adr:(Vnn vnn)</code>	<code>*</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 {<address>} {<method name>} {<method variant>}</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 {<address>} {<method name>} {<method variant>} {<true code>} {<false code>}</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:Vnnn</code>	<code>\object_new_cmethod:nnnn</code> $\{\langle address \rangle\}$ $\{\langle method\ name \rangle\}$ $\{\langle method\ arguments \rangle\}$ $\{\langle code \rangle\}$ 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 <code>\cs_new:Nn</code> . From: 2.0
--	--

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

Calls the specified method. This function is expandable if and only if the specified method was not declared `protected`.
 From: 2.0

4.4 Constant member creation

Unlike normal variables, constant variables in L^AT_EX3 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_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>	<code>\object_newconst_<type>:nnn</code> $\{\langle address \rangle\}$ $\{\langle constant\ name \rangle\}$ $\{\langle value \rangle\}$ Creates a constant variable with type $\langle type \rangle$ and sets its value to $\langle value \rangle$. From: 1.1
--	--

<code>\object_newconst_seq_from_clist:nnn</code> <code>\object_newconst_seq_from_clist:Vnn</code>	<code>\object_newconst_seq_from_clist:nnn</code> $\{\langle address \rangle\}$ $\{\langle constant\ name \rangle\}$ $\{\langle comma-list \rangle\}$
--	---

Creates a `seq` constant which is set to contain all the items in $\langle comma-list \rangle$.
 From: 1.1

<code>\object_newconst_prop_from_keyval:nnn</code> <code>\object_newconst_prop_from_keyval:Vnn</code>	<code>\object_newconst_prop_from_keyval:nnn</code> $\{\langle address \rangle\}$ $\{\langle constant\ name \rangle\}$ $\{$ $\langle key \rangle = \langle value \rangle, \dots$ $\}$
--	---

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

4.5 Macros

<code>\object_macro_adr:nn</code> *	<code>\object_macro_adr:nn {⟨address⟩} {⟨macro name⟩}</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 {⟨address⟩} {⟨macro name⟩}</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.

4.6 Proxy utilities and object creation

<code>\object_if_proxy_p:n</code> *	<code>\object_if_proxy_p:n {⟨address⟩}</code>
<code>\object_if_proxy_p:V</code> *	<code>\object_if_proxy:nTF {⟨address⟩} {⟨true code⟩} {⟨false code⟩}</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

<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^ATeX before 2019) it'll require slow processing. Don't use it in speed critical parts, instead use `\object_test_proxy:nN`.

From: 2.0

<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:VNNTF</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

<u>\c_proxy_address_str</u>	The address of the proxy object in the rawobjects module. From: 1.0
<u>\object_create:nnnNN</u> <u>\object_create:VnnNN</u>	\object_create:nnnNN {<proxy address>} {<module>} {<id>} <scope> <visibility> Creates an object by using the proxy at <proxy address> and the specified parameters. From: 1.0
<u>\embedded_create:nnn</u> <u>\embedded_create:(Vnn nvn)</u>	\embedded_create:nnn {<parent object>} {<proxy address>} {<id>} Creates an embedded object with name <id> inside <parent object>. From: 2.2
<u>\c_object_local_str</u> <u>\c_object_global_str</u>	Possible values for <scope> parameter. From: 1.0
<u>\c_object_public_str</u> <u>\c_object_private_str</u>	Possible values for <visibility> parameter. From: 1.0
<u>\object_create_set:NnnnNN</u> <u>\object_create_set:(NVnnNN NnnfNN)</u> <u>\object_create_gset:NnnnNN</u> <u>\object_create_gset:(NVnnNN NnnfNN)</u>	\object_create_set:NnnnNN <str var> {<proxy address>} {<module>} {<id>} <scope> <visibility> Creates an object and sets its fully expanded address inside <str var>. From: 1.0
<u>\object_allocate_incr:NNnnNN</u> <u>\object_allocate_incr:NNVnNN</u> <u>\object_gallocate_incr:NNnnNN</u> <u>\object_allocate_gincr:NNnnNN</u> <u>\object_allocate_gincr:NNVnNN</u> <u>\object_gallocate_gincr:NNnnNN</u> <u>\object_gallocate_gincr:NNVnNN</u>	\object_allocate_incr:NNnnNN <str var> <int var> {<proxy address>} {<module>} <scope> <visibility> Build a new object address with module <module> and an identifier generated from <proxy address> and the integer contained inside <int var>, then increments <int var>. This is very useful when you need to create a lot of objects, each of them on a different address. the _incr version increases <int var> locally whereas _gincr does it globally. From: 1.1
<u>\proxy_create:nnN</u> <u>\proxy_create_set:NnnN</u> <u>\proxy_create_gset:NnnN</u>	\proxy_create:nnN {<module>} {<id>} <visibility> \proxy_create_set:NnnN <str var> {<module>} {<id>} <visibility> Creates a global proxy object. From: 1.0

<hr/> <code>\proxy_push_member:nnn</code> <code>\proxy_push_member:Vnn</code> <hr/>	<code>\proxy_push_member:nnn {<proxy address>} {<member name>} {<member type>}</code> 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 <code>\object_member_type</code> functions. From: 1.0
<hr/> <code>\proxy_push_embedded:nnn</code> <code>\proxy_push_embedded:Vnn</code> <hr/>	<code>\proxy_push_embedded:nnn {<proxy address>} {<embedded object name>} {<embedded object proxy>}</code> Updates a proxy object with a new embedded object specification. From: 2.2
<hr/> <code>\object_assign:nn</code> <code>\object_assign:(Vn nV VV)</code> <hr/>	<code>\object_assign:nn {<to address>} {<from address>}</code> Assigns the content of each variable of object at <code><from address></code> to each corresponsive variable in <code><to address></code> . Both the objects should be created with the same proxy object and only variables listed in the proxy are assigned. From: 1.0

5 Examples

Example 1

Create a public proxy with id `myproxy` with the specification of a single member variable with name `myvar` and type `t1`, 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 }{ t1 }
```

Then create a new object with name `myobj` with that proxy, assign then token list `\c_dollar_str{}` ~ `dollar` ~ `\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

6 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, ...

7 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 {loc}
4 \str_const:Nn \c_object_global_str {glo}
5 \str_const:Nn \c_object_public_str {pub}
6 \str_const:Nn \c_object_private_str {pri}
7
8 \str_const:Nn \c__rawobjects_const_str {con}

```

(End definition for `\c_object_local_str` and others. These variables are documented on page 11.)

`\object_address:nn` Get address of an object

```

9 \cs_new:Nn \object_address:nn {
10   \tl_to_str:n { #1 _ #2 }
11 }

```

(End definition for `\object_address:nn`. This function is documented on page 5.)

`\object_embedded_adr:nn` Address of embedded object

```

12
13 \cs_new:Nn \object_embedded_adr:nn
14   {
15     #1 \tl_to_str:n{ _SUB_ #2 }
16   }
17
18 \cs_generate_variant:Nn \object_embedded_adr:nn{ Vn }
19

```

(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
20
21 \cs_new_protected:Nn \object_address_set:Nnn {
22   \str_set:Nn #1 { #2 _ #3 }
23 }
24
25 \cs_new_protected:Nn \object_address_gset:Nnn {
26   \str_gset:Nn #1 { #2 _ #3 }
27 }
28

```

(End definition for `\object_address_set:Nnn` and `\object_address_gset:Nnn`. These functions are documented on page 5.)

```

29 \cs_new:Nn \__rawobjects_object_modvar:n{
30   c __ #1 _ MODULE _ str
31 }
32
33 \cs_new:Nn \__rawobjects_object_pxyvar:n{
34   c __ #1 _ PROXY _ str
35 }
36

```

```

37 \cs_new:Nn \__rawobjects_object_scovar:n{
38   c __ #1 _ SCOPE _ str
39 }
40
41 \cs_new:Nn \__rawobjects_object_visvar:n{
42   c __ #1 _ VISIB _ str
43 }
44
45 \cs_generate_variant:Nn \__rawobjects_object_modvar:n { V }
46 \cs_generate_variant:Nn \__rawobjects_object_pxyvar:n { V }
47 \cs_generate_variant:Nn \__rawobjects_object_scovar:n { V }
48 \cs_generate_variant:Nn \__rawobjects_object_visvar:n { V }

```

`\object_if_exist_p:n` Tests if object exists.

`\object_if_exist:nTF`

```

49
50 \prg_new_conditional:Nnn \object_if_exist:n { p, T, F, TF }
51 {
52   \cs_if_exist:cTF
53   {
54     \__rawobjects_object_modvar:n { #1 }
55   }
56   {
57     \prg_return_true:
58   }
59   {
60     \prg_return_false:
61   }
62 }
63
64 \prg_generate_conditional_variant:Nnn \object_if_exist:n { V }
65 { p, T, F, TF }
66

```

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

`\object_get_module:n` Retrieve the name, module and generating proxy of an object

`\object_get_proxy_adr:n`

```

67 \cs_new:Nn \object_get_module:n {
68   \str_use:c { \__rawobjects_object_modvar:n { #1 } }
69 }
70 \cs_new:Nn \object_get_proxy_adr:n {
71   \str_use:c { \__rawobjects_object_pxyvar:n { #1 } }
72 }
73
74 \cs_generate_variant:Nn \object_get_module:n { V }
75 \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 6.)

`\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`

```

76 \prg_new_conditional:Nnn \object_if_local:n {p, T, F, TF}
77 {
78   \str_if_eq:cNTF { \__rawobjects_object_scovar:n {#1} }
79   \c_object_local_str

```



```

80     {
81         \prg_return_true:
82     }
83     {
84         \prg_return_false:
85     }
86 }
87
88 \prg_new_conditional:Nnn \object_if_global:n {p, T, F, TF}
89 {
90     \str_if_eq:cNTF { \__rawobjects_object_scovar:n {#1} }
91     \c_object_global_str
92     {
93         \prg_return_true:
94     }
95     {
96         \prg_return_false:
97     }
98 }
99
100 \prg_new_conditional:Nnn \object_if_public:n {p, T, F, TF}
101 {
102     \str_if_eq:cNTF { \__rawobjects_object_visvar:n { #1 } }
103     \c_object_public_str
104     {
105         \prg_return_true:
106     }
107     {
108         \prg_return_false:
109     }
110 }
111
112 \prg_new_conditional:Nnn \object_if_private:n {p, T, F, TF}
113 {
114     \str_if_eq:cNTF { \__rawobjects_object_visvar:n {#1} }
115     \c_object_private_str
116     {
117         \prg_return_true:
118     }
119     {
120         \prg_return_false:
121     }
122 }
123
124 \prg_generate_conditional_variant:Nnn \object_if_local:n { V }
125 { p, T, F, TF }
126 \prg_generate_conditional_variant:Nnn \object_if_global:n { V }
127 { p, T, F, TF }
128 \prg_generate_conditional_variant:Nnn \object_if_public:n { V }
129 { p, T, F, TF }
130 \prg_generate_conditional_variant:Nnn \object_if_private:n { V }
131 { p, T, F, TF }

```

(End definition for \object_if_local:nTF and others. These functions are documented on page 6.)

`\object_macro_adr:nn` Generic macro address

`\object_macro_use:nn`

```
132
133 \cs_new:Nn \object_macro_adr:nn
134 {
135     #1 \tl_to_str:n{ _MACRO_ #2 }
136 }
137
138 \cs_generate_variant:Nn \object_macro_adr:nn{ Vn }
139
140 \cs_new:Nn \object_macro_use:nn
141 {
142     \use:c
143     {
144         \object_macro_adr:nn{ #1 }{ #2 }
145     }
146 }
147
148 \cs_generate_variant:Nn \object_macro_use:nn{ Vn }
149
```

(End definition for `\object_macro_adr:nn` and `\object_macro_use:nn`. These functions are documented on page 10.)

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

`\object_member_adr:nn`

```
150
151 \cs_new:Nn \__rawobjects_scope:n
152 {
153     \object_if_local:nTF { #1 }
154     {
155         1
156     }
157     {
158         \str_if_eq:cNTF { \__rawobjects_object_scovar:n { #1 } }
159         \c__rawobjects_const_str
160         {
161             c
162         }
163         {
164             g
165         }
166     }
167 }
168
169 \cs_new:Nn \__rawobjects_scope_pfx:n
170 {
171     \object_if_local:nF { #1 }
172     { g }
173 }
174
175 \cs_new:Nn \__rawobjects_vis_var:n
176 {
177     \object_if_private:nTF { #1 }
178     {
179         --

```

```

180     }
181     {
182     } -
183     }
184 }
185
186 \cs_new:Nn \__rawobjects_vis_fun:n
187 {
188     \object_if_private:nT { #1 }
189     {
190     } --
191     }
192 }
193
194 \cs_new:Nn \object_member_adr:nnn
195 {
196     \__rawobjects_scope:n { #1 }
197     \__rawobjects_vis_var:n { #1 }
198     #1 \tl_to_str:n { _ MEMBER _ #2 _ #3 }
199 }
200
201 \cs_generate_variant:Nn \object_member_adr:nnn { Vnn, vnn, nnv }
202
203 \cs_new:Nn \object_member_adr:nn
204 {
205     \object_member_adr:nnv { #1 } { #2 }
206     {
207         \object_rcmember_adr:nnn { #1 }
208         { #2 _ type } { str }
209     }
210 }
211
212 \cs_generate_variant:Nn \object_member_adr:nn { Vn }
213

```

(End definition for `\object_member_adr:nnn` and `\object_member_adr:nn`. These functions are documented on page 6.)

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

```

214
215 \cs_new:Nn \object_member_type:nn
216 {
217     \object_rcmember_use:nnn { #1 }
218     { #2 _ type } { str }
219 }
220

```

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

```

221
222 \msg_new:nnnn { rawobjects } { scoperr } { Nonstandard ~ scope }
223 {
224     Operation ~ not ~ permitted ~ on ~ object ~ #1 ~
225     ~ since ~ it ~ wasn't ~ declared ~ local ~ or ~ global
226 }

```

```

227
228 \cs_new_protected:Nn \__rawobjects_force_scope:n
229 {
230   \bool_if:nF
231   {
232     \object_if_local_p:n { #1 } || \object_if_global_p:n { #1 }
233   }
234   {
235     \msg_error:nnx { rawobjects }{ scoperr }{ #1 }
236   }
237 }
238

```

`\object_member_if_exist_p:nnn` Tests if the specified member exists

```

\object_member_if_exist:nnnTF
239
\object_member_if_exist_p:nn
240 \prg_new_conditional:Nnn \object_member_if_exist:nnn {p, T, F, TF }
\object_member_if_exist:nnTF
241 {
242   \cs_if_exist:cTF
243   {
244     \object_member_adr:nnn { #1 }{ #2 }{ #3 }
245   }
246   {
247     \prg_return_true:
248   }
249   {
250     \prg_return_false:
251   }
252 }
253
254 \prg_new_conditional:Nnn \object_member_if_exist:nn {p, T, F, TF }
255 {
256   \cs_if_exist:cTF
257   {
258     \object_member_adr:nn { #1 }{ #2 }
259   }
260   {
261     \prg_return_true:
262   }
263   {
264     \prg_return_false:
265   }
266 }
267
268 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nnn
269 { Vnn }{ p, T, F, TF }
270 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nn
271 { Vn }{ p, T, F, TF }
272

```

(End definition for `\object_member_if_exist:nnnTF` and `\object_member_if_exist:nnTF`. These functions are documented on page 7.)

`\object_new_member:nnn` Creates a new member variable

```

273

```

```

274 \cs_new_protected:Nn \object_new_member:nnn
275 {
276   \__rawobjects_force_scope:n { #1 }
277   \cs_if_exist_use:cT { #3 _ new:c }
278   {
279     { \object_member_adr:nnn { #1 } { #2 } { #3 } }
280   }
281 }
282
283 \cs_generate_variant:Nn \object_new_member:nnn { Vnn, nnv }
284

```

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

`\object_member_use:nnn` Uses a member variable
`\object_member_use:nn`

```

285
286 \cs_new:Nn \object_member_use:nnn
287 {
288   \cs_if_exist_use:cT { #3 _ use:c }
289   {
290     { \object_member_adr:nnn { #1 } { #2 } { #3 } }
291   }
292 }
293
294 \cs_new:Nn \object_member_use:nn
295 {
296   \object_member_use:nnv { #1 } { #2 }
297   {
298     \object_rcmember_adr:nnn { #1 }
299     { #2 _ type } { str }
300   }
301 }
302
303 \cs_generate_variant:Nn \object_member_use:nnn { Vnn, vnn, nnv }
304 \cs_generate_variant:Nn \object_member_use:nn { Vn }
305

```

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

`\object_member_set:nnnn` Set the value a member.
`\object_member_set_eq:nnn`

```

306
307 \cs_new_protected:Nn \object_member_set:nnnn
308 {
309   \__rawobjects_force_scope:n { #1 }
310   \cs_if_exist_use:cT
311   {
312     #3 _ \__rawobjects_scope_pfx:n { #1 } set:cn
313   }
314   {
315     { \object_member_adr:nnn { #1 } { #2 } { #3 } } { #4 }
316   }
317 }
318
319 \cs_generate_variant:Nn \object_member_set:nnnn { Vnnn, nnvn }

```

```

320
321 \cs_new_protected:Nn \object_member_set:nnn
322 {
323   \object_member_set:nnvn { #1 } { #2 }
324   {
325     \object_rcmember_adr:nnn { #1 }
326     { #2 _ type } { str }
327   } { #3 }
328 }
329
330 \cs_generate_variant:Nn \object_member_set:nnn { Vnn }
331

```

(End definition for `\object_member_set:nnnn` and `\object_member_set_eq:nnn`. These functions are documented on page 7.)

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

Make a member equal to another variable.

```

332
333 \cs_new_protected:Nn \object_member_set_eq:nnnN
334 {
335   \__rawobjects_force_scope:n { #1 }
336   \cs_if_exist_use:cT
337   {
338     #3 _ \__rawobjects_scope_pfx:n { #1 } set _ eq:cN
339   }
340   {
341     { \object_member_adr:nnn { #1 } { #2 } { #3 } } #4
342   }
343 }
344
345 \cs_generate_variant:Nn \object_member_set_eq:nnnN { VnnN, nnc, Vnnc, nnvN }
346
347 \cs_new_protected:Nn \object_member_set_eq:nnN
348 {
349   \object_member_set_eq:nnvN { #1 } { #2 }
350   {
351     \object_rcmember_adr:nnn { #1 }
352     { #2 _ type } { str }
353   } #3
354 }
355
356 \cs_generate_variant:Nn \object_member_set_eq:nnN { VnN, nnc, Vnc }
357

```

(End definition for `\object_member_set_eq:nnnN` and `\object_member_set_eq:nnN`. These functions are documented on page 7.)

`\object_ncmember_adr:nnn`
`\object_rcmember_adr:nnn`

Get the address of a near/remote constant.

```

358
359 \cs_new:Nn \object_ncmember_adr:nnn
360 {
361   c _ #1 \tl_to_str:n { _ CONST _ #2 _ #3 }
362 }
363
364 \cs_generate_variant:Nn \object_ncmember_adr:nnn { Vnn, vnn }

```

```

365
366 \cs_new:Nn \object_rcmember_adr:nnn
367 {
368   \object_ncmember_adr:vnn { \__rawobjects_object_pxyvar:n { #1 } }
369   { #2 } { #3 }
370 }
371
372 \cs_generate_variant:Nn \object_rcmember_adr:nnn { Vnn }

```

(End definition for \object_ncmember_adr:nnn and \object_rcmember_adr:nnn. These functions are documented on page 8.)

\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
373
374 \prg_new_conditional:Nnn \object_ncmember_if_exist:nnn {p, T, F, TF }
375 {
376   \cs_if_exist:cTF
377   {
378     \object_ncmember_adr:nnn { #1 } { #2 } { #3 }
379   }
380   {
381     \prg_return_true:
382   }
383   {
384     \prg_return_false:
385   }
386 }
387
388 \prg_new_conditional:Nnn \object_rcmember_if_exist:nnn {p, T, F, TF }
389 {
390   \cs_if_exist:cTF
391   {
392     \object_rcmember_adr:nnn { #1 } { #2 } { #3 }
393   }
394   {
395     \prg_return_true:
396   }
397   {
398     \prg_return_false:
399   }
400 }
401
402 \prg_generate_conditional_variant:Nnn \object_ncmember_if_exist:nnn
403 { Vnn } { p, T, F, TF }
404 \prg_generate_conditional_variant:Nnn \object_rcmember_if_exist:nnn
405 { Vnn } { p, T, F, TF }
406

```

(End definition for \object_ncmember_if_exist:nnnTF and \object_rcmember_if_exist:nnnTF. These functions are documented on page 8.)

\object_ncmember_use:nnn Uses a near/remote constant.

```

\object_rcmember_use:nnn
407
408 \cs_new:Nn \object_ncmember_use:nnn
409 {

```

```

410     \cs_if_exist_use:cT { #3 _ use:c }
411     {
412         { \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 } }
413     }
414 }
415
416 \cs_new:Nn \object_rcmember_use:nnn
417 {
418     \cs_if_exist_use:cT { #3 _ use:c }
419     {
420         { \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 } }
421     }
422 }
423
424 \cs_generate_variant:Nn \object_ncmember_use:nnn { Vnn }
425 \cs_generate_variant:Nn \object_rcmember_use:nnn { Vnn }
426

```

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

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

```

427
428 \cs_new_protected:Nn \object_newconst:nnnn
429 {
430     \use:c { #3 _ const:cn }
431     {
432         \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 }
433     }
434     { #4 }
435 }
436

```

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

`\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`

```

437
438 \cs_new_protected:Nn \object_newconst_tl:nnn
439 {
440     \object_newconst:nnnn { #1 }{ #2 }{ tl }{ #3 }
441 }
442 \cs_new_protected:Nn \object_newconst_str:nnn
443 {
444     \object_newconst:nnnn { #1 }{ #2 }{ str }{ #3 }
445 }
446 \cs_new_protected:Nn \object_newconst_int:nnn
447 {
448     \object_newconst:nnnn { #1 }{ #2 }{ int }{ #3 }
449 }
450 \cs_new_protected:Nn \object_newconst_clist:nnn
451 {
452     \object_newconst:nnnn { #1 }{ #2 }{ clist }{ #3 }
453 }
454 \cs_new_protected:Nn \object_newconst_dim:nnn
455 {

```



```

456     \object_newconst:nnnn { #1 }{ #2 }{ dim }{ #3 }
457   }
458 \cs_new_protected:Nn \object_newconst_skip:nnn
459 {
460     \object_newconst:nnnn { #1 }{ #2 }{ skip }{ #3 }
461   }
462 \cs_new_protected:Nn \object_newconst_fp:nnn
463 {
464     \object_newconst:nnnn { #1 }{ #2 }{ fp }{ #3 }
465   }
466
467 \cs_generate_variant:Nn \object_newconst_tl:nnn { Vnn }
468 \cs_generate_variant:Nn \object_newconst_str:nnn { Vnn }
469 \cs_generate_variant:Nn \object_newconst_int:nnn { Vnn }
470 \cs_generate_variant:Nn \object_newconst_clist:nnn { Vnn }
471 \cs_generate_variant:Nn \object_newconst_dim:nnn { Vnn }
472 \cs_generate_variant:Nn \object_newconst_skip:nnn { Vnn }
473 \cs_generate_variant:Nn \object_newconst_fp:nnn { Vnn }
474

```

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

`\object_newconst_seq_from_clist:nnn` Creates a `seq` constant.

```

475
476 \cs_new_protected:Nn \object_newconst_seq_from_clist:nnn
477 {
478     \seq_const_from_clist:cn
479     {
480         \object_ncmember_adr:nnn { #1 }{ #2 }{ seq }
481       }
482     { #3 }
483   }
484
485 \cs_generate_variant:Nn \object_newconst_seq_from_clist:nnn { Vnn }
486

```

(End definition for `\object_newconst_seq_from_clist:nnn`. This function is documented on page 9.)

`\object_newconst_prop_from_keyval:nnn` Creates a `prop` constant.

```

487
488 \cs_new_protected:Nn \object_newconst_prop_from_keyval:nnn
489 {
490     \prop_const_from_keyval:cn
491     {
492         \object_ncmember_adr:nnn { #1 }{ #2 }{ prop }
493       }
494     { #3 }
495   }
496
497 \cs_generate_variant:Nn \object_newconst_prop_from_keyval:nnn { Vnn }
498

```

(End definition for `\object_newconst_prop_from_keyval:nnn`. This function is documented on page 9.)

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

`\object_rcmethod_adr:nnn`

```
499
500 \cs_new:Nn \object_ncmethod_adr:nnn
501 {
502   #1 \tl_to_str:n { _ CMETHOD _ #2 : #3 }
503 }
504
505 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
506
507 \cs_new:Nn \object_rcmethod_adr:nnn
508 {
509   \object_ncmethod_adr:vnn
510   {
511     \__rawobjects_object_pxyvar:n { #1 }
512   }
513   { #2 }{ #3 }
514 }
515
516 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
517 \cs_generate_variant:Nn \object_rcmethod_adr:nnn { Vnn }
518
```

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

`\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`

```
519
520 \prg_new_conditional:Nnn \object_ncmethod_if_exist:nnn {p, T, F, TF }
521 {
522   \cs_if_exist:cTF
523   {
524     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
525   }
526   {
527     \prg_return_true:
528   }
529   {
530     \prg_return_false:
531   }
532 }
533
534 \prg_new_conditional:Nnn \object_rcmethod_if_exist:nnn {p, T, F, TF }
535 {
536   \cs_if_exist:cTF
537   {
538     \object_rcmethodr_adr:nnn { #1 }{ #2 }{ #3 }
539   }
540   {
541     \prg_return_true:
542   }
543   {
544     \prg_return_false:
545   }
546 }
```

```

547
548 \prg_generate_conditional_variant:Nnn \object_ncmethod_if_exist:nnn
549   { Vnn }{ p, T, F, TF }
550 \prg_generate_conditional_variant:Nnn \object_rcmethod_if_exist:nnn
551   { Vnn }{ p, T, F, TF }
552

```

(End definition for \object_ncmethod_if_exist:nnnTF and \object_rcmethod_if_exist:nnnTF. These functions are documented on page 8.)

\object_new_cmethod:nnnn Creates a new method

```

553
554 \cs_new_protected:Nn \object_new_cmethod:nnnn
555   {
556     \cs_new:cn
557     {
558       \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
559     }
560     { #4 }
561   }
562
563 \cs_generate_variant:Nn \object_new_cmethod:nnnn { Vnnn }
564

```

(End definition for \object_new_cmethod:nnnn. This function is documented on page 9.)

\object_ncmethod_call:nnn Calls the specified method.

\object_rcmethod_call:nnn

```

565
566 \cs_new:Nn \object_ncmethod_call:nnn
567   {
568     \use:c
569     {
570       \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
571     }
572   }
573
574 \cs_new:Nn \object_rcmethod_call:nnn
575   {
576     \use:c
577     {
578       \object_rcmethod_adr:nnn { #1 }{ #2 }{ #3 }
579     }
580   }
581
582 \cs_generate_variant:Nn \object_ncmethod_call:nnn { Vnn }
583 \cs_generate_variant:Nn \object_rcmethod_call:nnn { Vnn }
584

```

(End definition for \object_ncmethod_call:nnn and \object_rcmethod_call:nnn. These functions are documented on page 9.)

\c_proxy_address_str The address of the proxy object.

```

585 \str_const:Nx \c_proxy_address_str
586   { \object_address:nn { rawobjects }{ proxy } }

```

(End definition for \c_proxy_address_str. This variable is documented on page 11.)

Source of proxy object

```

587
588 \str_const:cn { \__rawobjects_object_modvar:V \c_proxy_address_str }
589   { rawobjects }
590 \str_const:cV { \__rawobjects_object_pxyvar:V \c_proxy_address_str }
591   \c_proxy_address_str
592 \str_const:cV { \__rawobjects_object_scovar:V \c_proxy_address_str }
593   \__rawobjects_const_str
594 \str_const:cV { \__rawobjects_object_visvar:V \c_proxy_address_str }
595   \c_object_public_str
596
597 \seq_const_from_clist:cn
598   {
599     \object_member_adr:Vnn \c_proxy_address_str { varlist }{ seq }
600   }
601   { varlist, objlist }
602
603 \object_newconst_str:Vnn \c_proxy_address_str { varlist_type }{ seq }
604 \object_newconst_str:Vnn \c_proxy_address_str { objlist_type }{ seq }
605
606 \seq_const_from_clist:cn
607   {
608     \object_member_adr:Vnn \c_proxy_address_str { objlist }{ seq }
609   }
610   {}
611

```

\object_if_proxy:p:n Test if an object is a proxy.

\object_if_proxy:nTF

```

612
613 \prg_new_conditional:Nnn \object_if_proxy:n {p, T, F, TF}
614   {
615     \object_test_proxy:nNTF { #1 }
616     \c_proxy_address_str
617     {
618       \prg_return_true:
619     }
620     {
621       \prg_return_false:
622     }
623   }
624

```

(End definition for \object_if_proxy:nTF. This function is documented on page 10.)

\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

```

625
626 \prg_generate_conditional_variant:Nnn \str_if_eq:nn { ve }{ TF }
627
628 \prg_new_conditional:Nnn \object_test_proxy:nn {p, T, F, TF}
629   {
630     \str_if_eq:veTF { \__rawobjects_object_pxyvar:n { #1 } }
631     { #2 }
632     {

```

```

633         \prg_return_true:
634     }
635     {
636         \prg_return_false:
637     }
638 }
639
640 \prg_new_conditional:Nnn \object_test_proxy:nN {p, T, F, TF}
641 {
642     \str_if_eq:cNTF { \__rawobjects_object_pxyvar:n { #1 } }
643     #2
644     {
645         \prg_return_true:
646     }
647     {
648         \prg_return_false:
649     }
650 }
651
652 \prg_generate_conditional_variant:Nnn \object_test_proxy:nn
653 { Vn }{p, T, F, TF}
654 \prg_generate_conditional_variant:Nnn \object_test_proxy:nN
655 { VN }{p, T, F, TF}
656

```

(End definition for `\object_test_proxy:nnTF` and `\object_test_proxy:nNTF`. These functions are documented on page 10.)

`\object_create:nnnNN`
`\object_create_set:NnnnNN`
`\object_create_gset:NnnnNN`
`\embedded_create:nnn`

Creates an object from a proxy

```

657
658 \msg_new:nnn { aa }{ mess }{ #1 }
659
660 \msg_new:nnnn { rawobjects }{ notproxy }{ Fake ~ proxy }
661 {
662     Object ~ #1 ~ is ~ not ~ a ~ proxy.
663 }
664
665 \cs_new_protected:Nn \__rawobjects_force_proxy:n
666 {
667     \object_if_proxy:nF { #1 }
668     {
669         \msg_error:nnn { rawobjects }{ notproxy }{ #1 }
670     }
671 }
672
673 \cs_new_protected:Nn \__rawobjects_create_anon:nnnNN
674 {
675
676     \__rawobjects_force_proxy:n { #1 }
677
678     \str_const:cn { \__rawobjects_object_modvar:n { #2 } }{ #3 }
679     \str_const:cx { \__rawobjects_object_pxyvar:n { #2 } }{ #1 }
680     \str_const:cV { \__rawobjects_object_scovar:n { #2 } }{ #4 }
681     \str_const:cV { \__rawobjects_object_visvar:n { #2 } }{ #5 }

```

```

682
683 \seq_map_inline:cn
684 {
685   \object_member_adr:nnn { #1 }{ varlist }{ seq }
686 }
687 {
688   \object_new_member:nnv { #2 }{ ##1 }
689   {
690     \object_ncmember_adr:nnn { #1 }{ ##1 _ type }{ str }
691   }
692 }
693
694 \seq_map_inline:cn
695 {
696   \object_member_adr:nnn { #1 }{ objlist }{ seq }
697 }
698 {
699   \embedded_create:nnv
700   { #2 }
701   {
702     \object_ncmember_adr:nnn { #1 }{ ##1 _ proxy }{ str }
703   }
704   { ##1 }
705 }
706 }
707
708 \cs_generate_variant:Nn \__rawobjects_create_anon:nnnNN { nnvcc }
709
710 \cs_new_protected:Nn \object_create:nnnNN
711 {
712   \__rawobjects_create_anon:nnnNN { #1 }{ \object_address:nn { #2 }{ #3 } }
713   { #2 } #4 #5
714 }
715
716 \cs_new_protected:Nn \object_create_set:NnnnNN
717 {
718   \object_create:nnnNN { #2 }{ #3 }{ #4 } #5 #6
719   \str_set:Nx #1 { \object_address:nn { #3 }{ #4 } }
720 }
721
722 \cs_new_protected:Nn \object_create_gset:NnnnNN
723 {
724   \object_create:nnnNN { #2 }{ #3 }{ #4 } #5 #6
725   \str_gset:Nx #1 { \object_address:nn { #3 }{ #4 } }
726 }
727
728 \cs_new_protected:Nn \embedded_create:nnn
729 {
730   \__rawobjects_create_anon:nnvcc { #2 }
731   {
732     \object_embedded_adr:nn
733     { #1 }{ #3 }
734   }
735   {

```

```

736     \__rawobjects_object_modvar:n{ #1 }
737   }
738   {
739     \__rawobjects_object_scovar:n{ #1 }
740   }
741   {
742     \__rawobjects_object_visvar:n{ #1 }
743   }
744 }
745
746 \cs_generate_variant:Nn \object_create:nnnNN { VnnNN }
747 \cs_generate_variant:Nn \object_create_set:NnnnnNN { NVnnNN, NnnfNN }
748 \cs_generate_variant:Nn \object_create_gset:NnnnnNN { NVnnNN, NnnfNN }
749 \cs_generate_variant:Nn \embedded_create:nnn { nvN, Vnn }
750

```

(End definition for \object_create:nnnNN and others. These functions are documented on page 11.)

\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

```

751
752 \cs_new:Nn \__rawobjects_combine_aux:nnn
753 {
754   anon . #3 . #2 . #1
755 }
756
757 \cs_generate_variant:Nn \__rawobjects_combine_aux:nnn { Vnf }
758
759 \cs_new:Nn \__rawobjects_combine:Nn
760 {
761   \__rawobjects_combine_aux:Vnf #1 { #2 }
762   {
763     \cs_to_str:N #1
764   }
765 }
766
767 \cs_new_protected:Nn \object_allocate_incr:NNnnNN
768 {
769   \object_create_set:NnnfNN #1 { #3 }{ #4 }
770   {
771     \__rawobjects_combine:Nn #2 { #3 }
772   }
773   #5 #6
774
775   \int_incr:N #2
776 }
777
778 \cs_new_protected:Nn \object_gallocate_incr:NNnnNN
779 {
780   \object_create_gset:NnnfNN #1 { #3 }{ #4 }
781   {
782     \__rawobjects_combine:Nn #2 { #3 }
783   }
784   #5 #6
785

```

```

786     \int_incr:N #2
787   }
788
789   \cs_generate_variant:Nn \object_allocate_incr:NNnnNN { NNvNN }
790
791   \cs_generate_variant:Nn \object_gallocate_incr:NNnnNN { NNvNN }
792
793   \cs_new_protected:Nn \object_allocate_gincr:NNnnNN
794   {
795     \object_create_set:NnnfNN #1 { #3 }{ #4 }
796     {
797       \__rawobjects_combine:Nn #2 { #3 }
798     }
799     #5 #6
800
801     \int_gincr:N #2
802   }
803
804   \cs_new_protected:Nn \object_gallocate_gincr:NNnnNN
805   {
806     \object_create_gset:NnnfNN #1 { #3 }{ #4 }
807     {
808       \__rawobjects_combine:Nn #2 { #3 }
809     }
810     #5 #6
811
812     \int_gincr:N #2
813   }
814
815   \cs_generate_variant:Nn \object_allocate_gincr:NNnnNN { NNvNN }
816
817   \cs_generate_variant:Nn \object_gallocate_gincr:NNnnNN { NNvNN }
818

```

(End definition for \object_allocate_incr:NNnnNN and others. These functions are documented on page 11.)

```

\proxy_create:nnN Creates a new proxy object
\proxy_create_set:NnnN
\proxy_create_gset:NnnN
819
820 \cs_new_protected:Nn \proxy_create:nnN
821 {
822   \object_create:VnnNN \c_proxy_address_str { #1 }{ #2 }
823   \c_object_global_str #3
824 }
825
826 \cs_new_protected:Nn \proxy_create_set:NnnN
827 {
828   \object_create_set:NvnnNN #1 \c_proxy_address_str { #2 }{ #3 }
829   \c_object_global_str #4
830 }
831
832 \cs_new_protected:Nn \proxy_create_gset:NnnN
833 {
834   \object_create_gset:NvnnNN #1 \c_proxy_address_str { #2 }{ #3 }

```



```

835     \c_object_global_str #4
836   }
837

```

(End definition for `\proxy_create:nnN`, `\proxy_create_set:NnnN`, and `\proxy_create_gset:NnnN`. These functions are documented on page 11.)

`\proxy_push_member:nnn` Push a new member inside a proxy.

```

838 \cs_new_protected:Nn \proxy_push_member:nnn
839 {
840   \__rawobjects_force_scope:n { #1 }
841   \object_newconst_str:nnn { #1 }{ #2 _ type }{ #3 }
842   \seq_gput_left:cn
843   {
844     \object_member_adr:nnn { #1 }{ varlist }{ seq }
845   }
846   { #2 }
847 }
848
849 \cs_generate_variant:Nn \proxy_push_member:nnn { Vnn }
850

```

(End definition for `\proxy_push_member:nnn`. This function is documented on page 12.)

`\proxy_push_embedded:nnn` Push a new embedded object inside a proxy.

```

851
852 \cs_generate_variant:Nn \object_newconst_str:nnn { nnx }
853
854 \cs_new_protected:Nn \proxy_push_embedded:nnn
855 {
856   \__rawobjects_force_scope:n { #1 }
857   \object_newconst_str:nnx { #1 }{ #2 _ proxy }{ #3 }
858   \seq_gput_left:cn
859   {
860     \object_member_adr:nnn { #1 }{ objlist }{ seq }
861   }
862   { #2 }
863 }
864
865 \cs_generate_variant:Nn \proxy_push_embedded:nnn { Vnn }
866

```

(End definition for `\proxy_push_embedded:nnn`. This function is documented on page 12.)

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

```

867 \cs_new_protected:Nn \object_assign:nn
868 {
869   \seq_map_inline:cn
870   {
871     \object_member_adr:vnn
872     {
873       \__rawobjects_object_pxyvar:n { #1 }
874     }
875     { varlist }{ seq }
876   }

```

```

877     {
878       \object_member_set_eq:nnc { #1 }{ ##1 }
879       {
880         \object_member_adr:nn{ #2 }{ ##1 }
881       }
882     }
883   }
884
885   \cs_generate_variant:Nn \object_assign:nn { nV, Vn, VV }

```

(End definition for `\object_assign:nn`. This function is documented on page [12](#).)

A simple forward list proxy

```

886
887   \cs_new_protected:Nn \rawobjects_fwl_inst:n
888   {
889     \object_if_exist:nF
890     {
891       \object_address:nn { rawobjects }{ fwl ! #1 }
892     }
893     {
894       \proxy_create:nnN { rawobjects }{ fwl ! #1 } \c_object_private_str
895       \proxy_push_member
896       {
897         \object_address:nn { rawobjects }{ fwl ! #1 }
898       }
899       { next }{ str }
900     }
901   }
902
903   \cs_new_protected:Nn \rawobjects_fwl_newnode:nnnNN
904   {
905     \rawobjects_fwl_inst:n { #1 }
906     \object_create:nnnNN
907     {
908       \object_address:nn { rawobjects }{ fwl ! #1 }
909     }
910     { #2 }{ #3 } #4 #5
911   }
912
913   \</package>

```