

The lt3rawobjects package

Paolo De Donato

Released on 2022/11/27 Version 2.1.1

Contents

1	Introduction	1
2	To do	2
3	Objects and proxies	2
4	Library functions	3
4.1	Base object functions	3
4.2	Members	4
4.3	Methods	5
4.4	Constant member creation	6
4.5	Proxy utilities and object creation	7
5	Examples	9
6	Templated proxies	10
7	Implementation	11

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.

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 Instead changes introduced in minor and patch version updates are always backward compatible, and any withdrawn function is declared deprecated instead of being removed.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 To do

- Uniform declarations for templated proxies;
- Constant objects.

3 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*.

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 your 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 mutable members in each of your objects you can push their specifications inside the generating proxy via `\proxy_push_member`. In this way either object created from such proxy will have the specified members. 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 managment, 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_<category><item>_adr` where `<item>` is `member` or `method` and `<category>` is empty for mutable members, `nc` for near constants and `rc` for remote constants. For example `\object_rcmethod_adr` retrieves the address of specified remote constant method.

4 Library functions

4.1 Base object functions

<hr/> <code>\object_address:nn</code> \star	<code>\object_address:nn {<module>} {<id>}</code>	Composes the address of object in module <code><module></code> with identifier <code><id></code> and places it in the input stream. Notice that <code><module></code> and <code><id></code> 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 <code>V</code> , <code>f</code> or <code>e</code> variants. From: 1.0
<hr/> <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 <code><str var></code> . From: 1.1
<hr/> <code>\object_if_exist_p:n</code> \star <code>\object_if_exist_p:V</code> \star <code>\object_if_exist:nTF</code> \star <code>\object_if_exist:VTF</code> \star	<code>\object_if_exist_p:n {<address>}</code> <code>\object_if_exist:nTF {<address>} {<true code>} {<false code>}</code>	Tests if an object was instantiated at the specified address. From: 1.0
<hr/> <code>\object_get_module:n</code> \star <code>\object_get_module:V</code> \star <code>\object_get_proxy_adr:n</code> \star <code>\object_get_proxy_adr:V</code> \star	<code>\object_get_module:n {<address>}</code> <code>\object_get_proxy_adr:n {<address>}</code>	Get the object module and its generator. From: 1.0

```

\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 *

```

```

\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 *

```

4.2 Members

```

\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 *

```

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

```

\object_member_if_exist_p:nnn * \object_member_if_exist_p:nnn {\address} {\member name} {\member
\object_member_if_exist_p:Vnn * type}
\object_member_if_exist:nnnTF * \object_member_if_exist:nnnTF {\address} {\member name} {\member
\object_member_if_exist:VnnTF * type} {\true code} {\false code}
\object_member_if_exist_p:nn * \object_member_if_exist_p:nn {\address} {\member name}
\object_member_if_exist_p:Vn * \object_member_if_exist:nnTF {\address} {\member name} {\true code}
\object_member_if_exist:nnTF * {\false code}
\object_member_if_exist:VnTF *

```

Tests if the specified member exist.

From: 2.0

```

\object_member_type:nn * \object_member_type:nn {\address} {\member name}
\object_member_type:Vn *

```

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

```

\object_new_member:nnn * \object_new_member:nnn {\address} {\member name} {\member type}
\object_new_member:(Vnn|nnv) *

```

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⟩} {⟨valuer⟩}</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 Vnnnc)</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 nnnc Vnc)</code>	<code>{⟨variable⟩}</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 {⟨address⟩} {⟨member name⟩} {⟨member type⟩}</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 {⟨address⟩} {⟨member name⟩} {⟨member type⟩}</code>
<code>\object_ncmember_if_exist_p:Vnn</code>	<code>*</code>	<code>{⟨true code⟩} {⟨false code⟩}</code>
<code>\object_ncmember_if_exist:nnnTF</code>	<code>*</code>	<code>\object_ncmember_if_exist:nnnTF {⟨address⟩} {⟨member name⟩} {⟨member type⟩}</code>
<code>\object_ncmember_if_exist:VnnTF</code>	<code>*</code>	<code>{⟨true code⟩} {⟨false 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_ncmember_use:nnn</code>	<code>*</code>	<code>\object_ncmember_use:nnn {⟨address⟩} {⟨member name⟩} {⟨member type⟩}</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

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</code>
<code>\object_ncmethod_adr:(Vnn vnn)</code>	<code>*</code>	<code>variant})</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</code>
<code>\object_ncmethod_if_exist_p:Vnn</code>	<code>*</code>	<code>variant})</code>
<code>\object_ncmethod_if_exist:nnnTF</code>	<code>*</code>	<code>\object_ncmethod_if_exist:nnnTF {⟨address⟩} {⟨method name⟩} {⟨method</code>
<code>\object_ncmethod_if_exist:VnnTF</code>	<code>*</code>	<code>variant}) {⟨true code⟩} {⟨false 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_cmetho:nnnn</code>	<code>\object_new_cmetho:nnnn {⟨address⟩} {⟨method name⟩} {⟨method arguments⟩} {⟨code⟩}</code>
<code>\object_new_cmetho:Vnnn</code>	

Creates a new method with specified name and argument types. The `{⟨method arguments⟩}` 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 {⟨address⟩} {⟨method name⟩} {⟨method variant⟩}</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

4.4 Constant member creation

Unlike normal variables, constant variables in $\text{\LaTeX}3$ 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 {<address>} {<constant name>} {<value>}</code>
<code>\object_newconst_tl:Vnn</code>	Creates a constant variable with type <code><type></code> and sets its value to <code><value></code> .
<code>\object_newconst_str:nnn</code>	From: 1.1
<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_seq_from_clist:nnn</code>	<code>\object_newconst_seq_from_clist:nnn {<address>} {<constant name>}</code>
<code>\object_newconst_seq_from_clist:Vnn</code>	<code>{<comma-list>}</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 {<address>} {<constant name>}</code>
<code>\object_newconst_prop_from_keyval:Vnn</code>	<code>{</code> <code> <key> = <value>, ...</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 {<address>} {<constant name>} {<type>} {<value>}</code>
------------------------------------	---

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

4.5 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

<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^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

<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 proxy object in the <code>rawobjects</code> module.
	From: 1.0

<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>\c_object_local_str</code>	Possible values for <i>⟨scope⟩</i> parameter.
<code>\c_object_global_str</code>	From: 1.0

<hr/>	
<code>\c_object_public_str</code>	Possible values for <i>⟨visibility⟩</i> parameter.
<code>\c_object_private_str</code>	From: 1.0

<hr/>	
<code>\object_create_set:NnnnNN</code>	<code>\object_create_set:NnnnNN ⟨str var⟩ {⟨proxy address⟩} {⟨module⟩}</code>
<code>\object_create_set:(NVnnNN NnnfNN)</code>	<code>{⟨id⟩} ⟨scope⟩ ⟨visibility⟩</code>
<code>\object_create_gset:NnnnNN</code>	
<code>\object_create_gset:(NVnnNN NnnfNN)</code>	

Creates an object and sets its fully expanded address inside *⟨str var⟩*.
From: 1.0

<code>\object_allocate_incr:NNnnNN</code>	<code>\object_allocate_incr:NNnnNN <str var> <int var> {<proxy address>}</code>
<code>\object_allocate_incr:NNVnNN</code>	<code>{<module>} <scope> <visibility></code>
<code>\object_gallocate_incr:NNnnNN</code>	
<code>\object_gallocate_incr:NNVnNN</code>	
<code>\object_allocate_gincr:NNnnNN</code>	
<code>\object_allocate_gincr:NNVnNN</code>	
<code>\object_gallocate_gincr:NNnnNN</code>	
<code>\object_gallocate_gincr:NNVnNN</code>	

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

<code>\proxy_create:nnN</code>	<code>\proxy_create:nnN {<module>} {<id>} <visibility></code>
<code>\proxy_create_set:NnnN</code>	<code>\proxy_create_set:NnnN <str var> {<module>} {<id>} <visibility></code>
<code>\proxy_create_gset:NnnN</code>	

Creates a global proxy object.

From: 1.0

<code>\proxy_push_member:nnn</code>	<code>\proxy_push_member:nnn {<proxy address>} {< member name >} {< member type >}</code>
<code>\proxy_push_member:Vnn</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 `\object_member_type` functions.

From: 1.0

<code>\object_assign:nn</code>	<code>\object_assign:nn {<to address>} {<from address>}</code>
<code>\object_assign:(Vn nV VV)</code>	

Assigns the content of each variable of object at *<from address>* to each corresponsive 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

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 \$

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 8.)

```
\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 3.)

```
\object_address_set:Nnn Saves the address of an object into a string variable
\object_address_gset:Nnn
12
13 \cs_new_protected:Nn \object_address_set:Nnn {
14   \str_set:Nn #1 { #2 _ #3 }
15 }
16
```

```

17 \cs_new_protected:Nn \object_address_gset:Nnn {
18   \str_gset:Nn #1 { #2 _ #3 }
19 }
20

```

(End definition for \object_address_set:Nnn and \object_address_gset:Nnn. These functions are documented on page 3.)

```

21 \cs_new:Nn \__rawobjects_object_modvar:n{
22   c __ #1 _ MODULE _ str
23 }
24
25 \cs_new:Nn \__rawobjects_object_pxyvar:n{
26   c __ #1 _ PROXY _ str
27 }
28
29 \cs_new:Nn \__rawobjects_object_scovar:n{
30   c __ #1 _ SCOPE _ str
31 }
32
33 \cs_new:Nn \__rawobjects_object_visvar:n{
34   c __ #1 _ VISIB _ str
35 }
36
37 \cs_generate_variant:Nn \__rawobjects_object_modvar:n { V }
38 \cs_generate_variant:Nn \__rawobjects_object_pxyvar:n { V }
39 \cs_generate_variant:Nn \__rawobjects_object_scovar:n { V }
40 \cs_generate_variant:Nn \__rawobjects_object_visvar:n { V }

```

\object_if_exist_p:n Tests if object exists.

\object_if_exist:nTF

```

41
42 \prg_new_conditional:Nnn \object_if_exist:n { p, T, F, TF }
43 {
44   \cs_if_exist:cTF
45   {
46     \__rawobjects_object_modvar:n { #1 }
47   }
48   {
49     \prg_return_true:
50   }
51   {
52     \prg_return_false:
53   }
54 }
55
56 \prg_generate_conditional_variant:Nnn \object_if_exist:n { V }
57 { p, T, F, TF }
58

```

(End definition for \object_if_exist:nTF. This function is documented on page 3.)

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

\object_get_proxy_adr:n

```

59 \cs_new:Nn \object_get_module:n {
60   \str_use:c { \__rawobjects_object_modvar:n { #1 } }
61 }

```

```

62 \cs_new:Nn \object_get_proxy_adr:n {
63   \str_use:c { \__rawobjects_object_pxyvar:n { #1 } }
64 }
65
66 \cs_generate_variant:Nn \object_get_module:n { V }
67 \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 3.)

```

\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

68 \prg_new_conditional:Nnn \object_if_local:n {p, T, F, TF}
69 {
70   \str_if_eq:cNTF { \__rawobjects_object_scovar:n {#1} }
71   \c_object_local_str
72   {
73     \prg_return_true:
74   }
75   {
76     \prg_return_false:
77   }
78 }
79
80 \prg_new_conditional:Nnn \object_if_global:n {p, T, F, TF}
81 {
82   \str_if_eq:cNTF { \__rawobjects_object_scovar:n {#1} }
83   \c_object_global_str
84   {
85     \prg_return_true:
86   }
87   {
88     \prg_return_false:
89   }
90 }
91
92 \prg_new_conditional:Nnn \object_if_public:n {p, T, F, TF}
93 {
94   \str_if_eq:cNTF { \__rawobjects_object_visvar:n { #1 } }
95   \c_object_public_str
96   {
97     \prg_return_true:
98   }
99   {
100     \prg_return_false:
101   }
102 }
103
104 \prg_new_conditional:Nnn \object_if_private:n {p, T, F, TF}
105 {
106   \str_if_eq:cNTF { \__rawobjects_object_visvar:n {#1} }
107   \c_object_private_str
108   {
109     \prg_return_true:
110   }

```

```

111     {
112         \prg_return_false:
113     }
114 }
115
116 \prg_generate_conditional_variant:Nnn \object_if_local:n { V }
117 { p, T, F, TF }
118 \prg_generate_conditional_variant:Nnn \object_if_global:n { V }
119 { p, T, F, TF }
120 \prg_generate_conditional_variant:Nnn \object_if_public:n { V }
121 { p, T, F, TF }
122 \prg_generate_conditional_variant:Nnn \object_if_private:n { V }
123 { p, T, F, TF }

```

(End definition for \object_if_local:nTF and others. These functions are documented on page 4.)

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

```

124
125 \cs_new:Nn \__rawobjects_scope:n
126 {
127     \object_if_local:nTF { #1 }
128     {
129         1
130     }
131     {
132         \str_if_eq:cNTF { \__rawobjects_object_scovar:n { #1 } }
133         \c__rawobjects_const_str
134         {
135             c
136         }
137         {
138             g
139         }
140     }
141 }
142
143 \cs_new:Nn \__rawobjects_scope_pfx:n
144 {
145     \object_if_local:nF { #1 }
146     { g }
147 }
148
149 \cs_new:Nn \__rawobjects_vis_var:n
150 {
151     \object_if_private:nTF { #1 }
152     {
153         --
154     }
155     {
156         -
157     }
158 }
159
160 \cs_new:Nn \__rawobjects_vis_fun:n

```

```

161 {
162   \object_if_private:nT { #1 }
163   {
164     --
165   }
166 }
167
168 \cs_new:Nn \object_member_adr:nnn
169 {
170   \__rawobjects_scope:n { #1 }
171   \__rawobjects_vis_var:n { #1 }
172   #1 \tl_to_str:n { _ MEMBER _ #2 _ #3 }
173 }
174
175 \cs_generate_variant:Nn \object_member_adr:nnn { Vnn, vnn, nnv }
176
177 \cs_new:Nn \object_member_adr:nn
178 {
179   \object_member_adr:nnv { #1 } { #2 }
180   {
181     \object_rcmember_adr:nnn { #1 }
182     { #2 _ type } { str }
183   }
184 }
185
186 \cs_generate_variant:Nn \object_member_adr:nn { Vn }

```

(End definition for `\object_member_adr:nnn` and `\object_member_adr:nn`. These functions are documented on page 4.)

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

```

187
188 \cs_new:Nn \object_member_type:nn
189 {
190   \object_rcmember_use:nnn { #1 }
191   { #2 _ type } { str }
192 }
193

```

(End definition for `\object_member_type:nn`. This function is documented on page 4.)

```

194
195 \msg_new:nnnn { rawobjects } { scoperr } { Nonstandard ~ scope }
196 {
197   Operation ~ not ~ permitted ~ on ~ object ~ #1 ~
198   ~ since ~ it ~ wasn't ~ declared ~ local ~ or ~ global
199 }
200
201 \cs_new_protected:Nn \__rawobjects_force_scope:n
202 {
203   \bool_if:nF
204   {
205     \object_if_local_p:n { #1 } || \object_if_global_p:n { #1 }
206   }
207   {

```

```

208         \msg_error:nnx { rawobjects }{ scoperr }{ #1 }
209     }
210 }
211

```

`\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`

```

212
213 \prg_new_conditional:Nnn \object_member_if_exist:nnn {p, T, F, TF }
214 {
215     \cs_if_exist:cTF
216     {
217         \object_member_adr:nnn { #1 }{ #2 }{ #3 }
218     }
219     {
220         \prg_return_true:
221     }
222     {
223         \prg_return_false:
224     }
225 }
226
227 \prg_new_conditional:Nnn \object_member_if_exist:nn {p, T, F, TF }
228 {
229     \cs_if_exist:cTF
230     {
231         \object_member_adr:nn { #1 }{ #2 }
232     }
233     {
234         \prg_return_true:
235     }
236     {
237         \prg_return_false:
238     }
239 }
240
241 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nnn
242 { Vnn }{ p, T, F, TF }
243 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nn
244 { Vn }{ p, T, F, TF }
245

```

(End definition for `\object_member_if_exist:nnnTF` and `\object_member_if_exist:nnTF`. These functions are documented on page 4.)

`\object_new_member:nnn`

Creates a new member variable

```

246
247 \cs_new_protected:Nn \object_new_member:nnn
248 {
249     \__rawobjects_force_scope:n { #1 }
250     \cs_if_exist_use:cT { #3 _ new:c }
251     {
252         \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
253     }
254 }

```



```

255
256 \cs_generate_variant:Nn \object_new_member:nnn { Vnn, nnv }
257

```

(End definition for `\object_new_member:nnn`. This function is documented on page 4.)

`\object_member_use:nnn` Uses a member variable

`\object_member_use:nn`

```

258
259 \cs_new:Nn \object_member_use:nnn
260 {
261   \cs_if_exist_use:cT { #3 _ use:c }
262   {
263     { \object_member_adr:nnn { #1 } { #2 } { #3 } }
264   }
265 }
266
267 \cs_new:Nn \object_member_use:nn
268 {
269   \object_member_use:nnv { #1 } { #2 }
270   {
271     \object_rcmember_adr:nnn { #1 }
272     { #2 _ type } { str }
273   }
274 }
275
276 \cs_generate_variant:Nn \object_member_use:nnn { Vnn, vnn, nnv }
277 \cs_generate_variant:Nn \object_member_use:nn { Vn }
278

```

(End definition for `\object_member_use:nnn` and `\object_member_use:nn`. These functions are documented on page 5.)

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

`\object_member_set_eq:nnn`

```

279
280 \cs_new_protected:Nn \object_member_set:nnnn
281 {
282   \__rawobjects_force_scope:n { #1 }
283   \cs_if_exist_use:cT
284   {
285     #3 _ \__rawobjects_scope_pfx:n { #1 } set:cn
286   }
287   {
288     { \object_member_adr:nnn { #1 } { #2 } { #3 } } { #4 }
289   }
290 }
291
292 \cs_generate_variant:Nn \object_member_set:nnnn { Vnnn, nnvn }
293
294 \cs_new_protected:Nn \object_member_set:nnn
295 {
296   \object_member_set:nnvn { #1 } { #2 }
297   {
298     \object_rcmember_adr:nnn { #1 }
299     { #2 _ type } { str }
300   } { #3 }

```

```

301 }
302
303 \cs_generate_variant:Nn \object_member_set:nnn { Vnn }
304

```

(End definition for `\object_member_set:nnnn` and `\object_member_set_eq:nnn`. These functions are documented on page 5.)

`\object_member_set_eq:nnnN` Make a member equal to another variable.

`\object_member_set_eq:nnN`

```

305
306 \cs_new_protected:Nn \object_member_set_eq:nnnN
307 {
308   \__rawobjects_force_scope:n { #1 }
309   \cs_if_exist_use:cT
310   {
311     #3 _ \__rawobjects_scope_pfx:n { #1 } set _ eq:cN
312   }
313   {
314     { \object_member_adr:nnn { #1 } { #2 } { #3 } } #4
315   }
316 }
317
318 \cs_generate_variant:Nn \object_member_set_eq:nnnN { VnnN, nnnC, VnnC, nnvN }
319
320 \cs_new_protected:Nn \object_member_set_eq:nnN
321 {
322   \object_member_set_eq:nnvN { #1 } { #2 }
323   {
324     \object_rcmember_adr:nnn { #1 }
325     { #2 _ type } { str }
326   } #3
327 }
328
329 \cs_generate_variant:Nn \object_member_set_eq:nnN { VnN, nnc, Vnc }
330

```

(End definition for `\object_member_set_eq:nnnN` and `\object_member_set_eq:nnN`. These functions are documented on page 5.)

`\object_ncmember_adr:nnn` Get the address of a near/remote constant.

`\object_rcmember_adr:nnn`

```

331
332 \cs_new:Nn \object_ncmember_adr:nnn
333 {
334   c _ #1 \tl_to_str:n { _ CONST _ #2 _ #3 }
335 }
336
337 \cs_generate_variant:Nn \object_ncmember_adr:nnn { Vnn, vnn }
338
339 \cs_new:Nn \object_rcmember_adr:nnn
340 {
341   \object_ncmember_adr:vnn { \__rawobjects_object_pxyvar:n { #1 } }
342   { #2 } { #3 }
343 }
344
345 \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 5.)

```

\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
346
347 \prg_new_conditional:Nnn \object_ncmember_if_exist:nnn {p, T, F, TF }
348 {
349   \cs_if_exist:cTF
350   {
351     \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 }
352   }
353   {
354     \prg_return_true:
355   }
356   {
357     \prg_return_false:
358   }
359 }
360
361 \prg_new_conditional:Nnn \object_rcmember_if_exist:nnn {p, T, F, TF }
362 {
363   \cs_if_exist:cTF
364   {
365     \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 }
366   }
367   {
368     \prg_return_true:
369   }
370   {
371     \prg_return_false:
372   }
373 }
374
375 \prg_generate_conditional_variant:Nnn \object_ncmember_if_exist:nnn
376 { Vnn }{ p, T, F, TF }
377 \prg_generate_conditional_variant:Nnn \object_rcmember_if_exist:nnn
378 { Vnn }{ p, T, F, TF }
379

```

(End definition for `\object_ncmember_if_exist:nnnTF` and `\object_rcmember_if_exist:nnnTF`. These functions are documented on page 5.)

```

\object_ncmember_use:nnn Uses a near/remote constant.
\object_rcmember_use:nnn
380
381 \cs_new:Nn \object_ncmember_use:nnn
382 {
383   \cs_if_exist_use:cT { #3 _ use:c }
384   {
385     \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 } }
386   }
387 }
388
389 \cs_new:Nn \object_rcmember_use:nnn
390 {

```

```

391     \cs_if_exist_use:cT { #3 _ use:c }
392     {
393         { \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 } }
394     }
395 }
396
397 \cs_generate_variant:Nn \object_ncmember_use:nnn { Vnn }
398 \cs_generate_variant:Nn \object_rcmember_use:nnn { Vnn }
399

```

(End definition for `\object_ncmember_use:nnn` and `\object_rcmember_use:nnn`. These functions are documented on page 5.)

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

```

400
401 \cs_new_protected:Nn \object_newconst:nnnn
402 {
403     \use:c { #3 _ const:cn }
404     {
405         \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 }
406     }
407     { #4 }
408 }
409

```

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

`\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`

```

410
411 \cs_new_protected:Nn \object_newconst_tl:nnn
412 {
413     \object_newconst:nnnn { #1 }{ #2 }{ tl }{ #3 }
414 }
415 \cs_new_protected:Nn \object_newconst_str:nnn
416 {
417     \object_newconst:nnnn { #1 }{ #2 }{ str }{ #3 }
418 }
419 \cs_new_protected:Nn \object_newconst_int:nnn
420 {
421     \object_newconst:nnnn { #1 }{ #2 }{ int }{ #3 }
422 }
423 \cs_new_protected:Nn \object_newconst_clist:nnn
424 {
425     \object_newconst:nnnn { #1 }{ #2 }{ clist }{ #3 }
426 }
427 \cs_new_protected:Nn \object_newconst_dim:nnn
428 {
429     \object_newconst:nnnn { #1 }{ #2 }{ dim }{ #3 }
430 }
431 \cs_new_protected:Nn \object_newconst_skip:nnn
432 {
433     \object_newconst:nnnn { #1 }{ #2 }{ skip }{ #3 }
434 }
435 \cs_new_protected:Nn \object_newconst_fp:nnn
436 {

```

```

437     \object_newconst:nnnn { #1 }{ #2 }{ fp }{ #3 }
438   }
439
440   \cs_generate_variant:Nn \object_newconst_tl:nnn { Vnn }
441   \cs_generate_variant:Nn \object_newconst_str:nnn { Vnn }
442   \cs_generate_variant:Nn \object_newconst_int:nnn { Vnn }
443   \cs_generate_variant:Nn \object_newconst_clist:nnn { Vnn }
444   \cs_generate_variant:Nn \object_newconst_dim:nnn { Vnn }
445   \cs_generate_variant:Nn \object_newconst_skip:nnn { Vnn }
446   \cs_generate_variant:Nn \object_newconst_fp:nnn { Vnn }
447

```

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

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

```

448
449   \cs_new_protected:Nn \object_newconst_seq_from_clist:nnn
450   {
451     \seq_const_from_clist:cn
452     {
453       \object_ncmember_adr:nnn { #1 }{ #2 }{ seq }
454     }
455     { #3 }
456   }
457
458   \cs_generate_variant:Nn \object_newconst_seq_from_clist:nnn { Vnn }
459

```

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

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

```

460
461   \cs_new_protected:Nn \object_newconst_prop_from_keyval:nnn
462   {
463     \prop_const_from_keyval:cn
464     {
465       \object_ncmember_adr:nnn { #1 }{ #2 }{ prop }
466     }
467     { #3 }
468   }
469
470   \cs_generate_variant:Nn \object_newconst_prop_from_keyval:nnn { Vnn }
471

```

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

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

`\object_rcmethod_adr:nnn`

```

472
473   \cs_new:Nn \object_ncmethod_adr:nnn
474   {
475     #1 \tl_to_str:n { _ CMETHOD _ #2 : #3 }
476   }
477
478   \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
479

```

```

480 \cs_new:Nn \object_rcmethod_adr:nnn
481 {
482   \object_ncmethod_adr:vnn
483   {
484     \__rawobjects_object_pxyvar:n { #1 }
485   }
486   { #2 }{ #3 }
487 }
488
489 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
490 \cs_generate_variant:Nn \object_rcmethod_adr:nnn { Vnn }
491

```

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

```

\object_ncmethod_if_exist:p:nnn
\object_ncmethod_if_exist:nnnTF
\object_rcmethod_if_exist:p:nnn
\object_rcmethod_if_exist:nnnTF

```

Tests if the specified member constant exists.

```

492
493 \prg_new_conditional:Nnn \object_ncmethod_if_exist:nnn {p, T, F, TF }
494 {
495   \cs_if_exist:cTF
496   {
497     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
498   }
499   {
500     \prg_return_true:
501   }
502   {
503     \prg_return_false:
504   }
505 }
506
507 \prg_new_conditional:Nnn \object_rcmethod_if_exist:nnn {p, T, F, TF }
508 {
509   \cs_if_exist:cTF
510   {
511     \object_rcmethodadr:nnn { #1 }{ #2 }{ #3 }
512   }
513   {
514     \prg_return_true:
515   }
516   {
517     \prg_return_false:
518   }
519 }
520
521 \prg_generate_conditional_variant:Nnn \object_ncmethod_if_exist:nnn
522 { Vnn }{ p, T, F, TF }
523 \prg_generate_conditional_variant:Nnn \object_rcmethod_if_exist:nnn
524 { Vnn }{ p, T, F, TF }
525

```

(End definition for `\object_ncmethod_if_exist:nnnTF` and `\object_rcmethod_if_exist:nnnTF`. These functions are documented on page 6.)

`\object_new_cmethod:nnnn` Creates a new method

```
526
527 \cs_new_protected:Nn \object_new_cmethod:nnnn
528 {
529   \cs_new:cn
530   {
531     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
532   }
533   { #4 }
534 }
535
536 \cs_generate_variant:Nn \object_new_cmethod:nnnn { Vnnn }
537
```

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

`\object_ncmethod_call:nnn` Calls the specified method.

`\object_rcmethod_call:nnn`

```
538
539 \cs_new:Nn \object_ncmethod_call:nnn
540 {
541   \use:c
542   {
543     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
544   }
545 }
546
547 \cs_new:Nn \object_rcmethod_call:nnn
548 {
549   \use:c
550   {
551     \object_rcmethod_adr:nnn { #1 }{ #2 }{ #3 }
552   }
553 }
554
555 \cs_generate_variant:Nn \object_ncmethod_call:nnn { Vnn }
556 \cs_generate_variant:Nn \object_rcmethod_call:nnn { Vnn }
557
```

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

`\c_proxy_address_str` The address of the proxy object.

```
558 \str_const:Nx \c_proxy_address_str
559 { \object_address:nn { rawobjects }{ proxy } }
```

(End definition for `\c_proxy_address_str`. This variable is documented on page 8.)

Source of proxy object

```
560 \str_const:cn { \__rawobjects_object_modvar:V \c_proxy_address_str }
561 { rawobjects }
562 \str_const:cV { \__rawobjects_object_pxyvar:V \c_proxy_address_str }
563 \c_proxy_address_str
564 \str_const:cV { \__rawobjects_object_scovar:V \c_proxy_address_str }
565 \c__rawobjects_const_str
566 \str_const:cV { \__rawobjects_object_visvar:V \c_proxy_address_str }
```

```

567 \c_object_public_str
568
569 \seq_const_from_clist:cn
570 {
571   \object_member_adr:Vnn \c_proxy_address_str { varlist }{ seq }
572 }
573 { varlist }
574
575 \object_newconst_str:Vnn \c_proxy_address_str { varlist_type }{ seq }
576

```

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

```

577
578 \prg_new_conditional:Nnn \object_if_proxy:n {p, T, F, TF}
579 {
580   \object_test_proxy:nNTF { #1 }
581   \c_proxy_address_str
582   {
583     \prg_return_true:
584   }
585   {
586     \prg_return_false:
587   }
588 }
589

```

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

\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

```

590
591 \prg_generate_conditional_variant:Nnn \str_if_eq:nn { ve }{ TF }
592
593 \prg_new_conditional:Nnn \object_test_proxy:nn {p, T, F, TF}
594 {
595   \str_if_eq:veTF { \__rawobjects_object_pxyvar:n { #1 } }
596   { #2 }
597   {
598     \prg_return_true:
599   }
600   {
601     \prg_return_false:
602   }
603 }
604
605 \prg_new_conditional:Nnn \object_test_proxy:nN {p, T, F, TF}
606 {
607   \str_if_eq:cNTF { \__rawobjects_object_pxyvar:n { #1 } }
608   #2
609   {
610     \prg_return_true:
611   }
612   {
613     \prg_return_false:
614   }

```



```

615 }
616
617 \prg_generate_conditional_variant:Nnn \object_test_proxy:nn
618 { Vn }{p, T, F, TF}
619 \prg_generate_conditional_variant:Nnn \object_test_proxy:nN
620 { VN }{p, T, F, TF}
621

```

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

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

Creates an object from a proxy

```

622
623 \msg_new:nnn { aa }{ mess }{ #1 }
624
625 \msg_new:nnnn { rawobjects }{ notproxy }{ Fake ~ proxy }
626 {
627   Object ~ #1 ~ is ~ not ~ a ~ proxy.
628 }
629
630 \cs_new_protected:Nn \__rawobjects_force_proxy:n
631 {
632   \object_if_proxy:nF { #1 }
633   {
634     \msg_error:nnn { rawobjects }{ notproxy }{ #1 }
635   }
636 }
637
638 \cs_new_protected:Nn \__rawobjects_create_anon:nnnNN
639 {
640
641   \__rawobjects_force_proxy:n { #1 }
642
643   \str_const:cn { \__rawobjects_object_modvar:n { #2 } }{ #3 }
644   \str_const:cx { \__rawobjects_object_pxyvar:n { #2 } }{ #1 }
645   \str_const:cV { \__rawobjects_object_scovar:n { #2 } }{ #4 }
646   \str_const:cV { \__rawobjects_object_visvar:n { #2 } }{ #5 }
647
648   \seq_map_inline:cn
649   {
650     \object_member_adr:nnn { #1 }{ varlist }{ seq }
651   }
652   {
653     \object_new_member:nnv { #2 }{ ##1 }
654     {
655       \object_ncmember_adr:nnn { #1 }{ ##1 _ type }{ str }
656     }
657   }
658 }
659
660 \cs_new_protected:Nn \object_create:nnnNN
661 {
662   \__rawobjects_create_anon:nnnNN { #1 }{ \object_address:nn { #2 }{ #3 } }
663   { #2 } #4 #5

```

```

664     }
665
666 \cs_new_protected:Nn \object_create_set:NnnnNN
667 {
668     \object_create:nnnNN { #2 } { #3 } { #4 } #5 #6
669     \str_set:Nx #1 { \object_address:nn { #3 } { #4 } }
670 }
671
672 \cs_new_protected:Nn \object_create_gset:NnnnNN
673 {
674     \object_create:nnnNN { #2 } { #3 } { #4 } #5 #6
675     \str_gset:Nx #1 { \object_address:nn { #3 } { #4 } }
676 }
677
678 \cs_generate_variant:Nn \object_create:nnnNN { VnnNN }
679 \cs_generate_variant:Nn \object_create_set:NnnnNN { NVnnNN, NnnfNN }
680 \cs_generate_variant:Nn \object_create_gset:NnnnNN { NVnnNN, NnnfNN }
681

```

(End definition for `\object_create:nnnNN`, `\object_create_set:NnnnNN`, and `\object_create_gset:NnnnNN`.
These functions are documented on page 8.)

\object_allocate_incr:NNnnNN Create an address and use it to instantiate an object

```

\object_allocate_incr:NNnnNN
\object_gallocate_incr:NNnnNN
\object_allocate_gincr:NNnnNN
\object_gallocate_gincr:NNnnNN
682
683 \cs_new:Nn \__rawobjects_combine_aux:nnn
684 {
685     anon . #3 . #2 . #1
686 }
687
688 \cs_generate_variant:Nn \__rawobjects_combine_aux:nnn { Vnf }
689
690 \cs_new:Nn \__rawobjects_combine:Nn
691 {
692     \__rawobjects_combine_aux:Vnf #1 { #2 }
693     {
694         \cs_to_str:N #1
695     }
696 }
697
698 \cs_new_protected:Nn \object_allocate_incr:NNnnNN
699 {
700     \object_create_set:NnnfNN #1 { #3 } { #4 }
701     {
702         \__rawobjects_combine:Nn #2 { #3 }
703     }
704     #5 #6
705
706     \int_incr:N #2
707 }
708
709 \cs_new_protected:Nn \object_gallocate_incr:NNnnNN
710 {
711     \object_create_gset:NnnfNN #1 { #3 } { #4 }
712     {

```

```

713     \__rawobjects_combine:Nn #2 { #3 }
714 }
715 #5 #6
716
717 \int_incr:N #2
718 }
719
720 \cs_generate_variant:Nn \object_allocate_incr:NNnnNN { NNVnNN }
721
722 \cs_generate_variant:Nn \object_gallocate_incr:NNnnNN { NNVnNN }
723
724 \cs_new_protected:Nn \object_allocate_gincr:NNnnNN
725 {
726   \object_create_set:NnnfNN #1 { #3 }{ #4 }
727   {
728     \__rawobjects_combine:Nn #2 { #3 }
729   }
730   #5 #6
731
732   \int_gincr:N #2
733 }
734
735 \cs_new_protected:Nn \object_gallocate_gincr:NNnnNN
736 {
737   \object_create_gset:NnnfNN #1 { #3 }{ #4 }
738   {
739     \__rawobjects_combine:Nn #2 { #3 }
740   }
741   #5 #6
742
743   \int_gincr:N #2
744 }
745
746 \cs_generate_variant:Nn \object_allocate_gincr:NNnnNN { NNVnNN }
747
748 \cs_generate_variant:Nn \object_gallocate_gincr:NNnnNN { NNVnNN }
749

```

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

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

Creates a new proxy object

```

750
751 \cs_new_protected:Nn \proxy_create:nnN
752 {
753   \object_create:VnnNN \c_proxy_address_str { #1 }{ #2 }
754   \c_object_global_str #3
755 }
756
757 \cs_new_protected:Nn \proxy_create_set:NnnN
758 {
759   \object_create_set:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
760   \c_object_global_str #4
761 }

```

```

762
763 \cs_new_protected:Nn \proxy_create_gset:NnnN
764 {
765   \object_create_gset:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
766   \c_object_global_str #4
767 }
768

```

(End definition for `\proxy_create:nnN`, `\proxy_create_set:NnnN`, and `\proxy_create_gset:NnnN`. These functions are documented on page 9.)

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

```

769 \cs_new_protected:Nn \proxy_push_member:nnn
770 {
771   \__rawobjects_force_scope:n { #1 }
772   \object_newconst_str:nnn { #1 }{ #2 _ type }{ #3 }
773   \seq_gput_left:cn
774   {
775     \object_member_adr:nnn { #1 }{ varlist }{ seq }
776   }
777   { #2 }
778 }
779
780 \cs_generate_variant:Nn \proxy_push_member:nnn { Vnn }
781

```

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

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

```

782 \cs_new_protected:Nn \object_assign:nn
783 {
784   \seq_map_inline:cn
785   {
786     \object_member_adr:vnn
787     {
788       \__rawobjects_object_pxyvar:n { #1 }
789     }
790     { varlist }{ seq }
791   }
792   {
793     \object_member_set_eq:nnc { #1 }{ ##1 }
794     {
795       \object_member_adr:nn{ #2 }{ ##1 }
796     }
797   }
798 }
799
800 \cs_generate_variant:Nn \object_assign:nn { nV, Vn, VV }

```

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

A simple forward list proxy

```

801
802 \cs_new_protected:Nn \rawobjects_fwl_inst:n
803 {
804   \object_if_exist:nF

```

```

805     {
806       \object_address:nn { rawobjects }{ fwl ! #1 }
807     }
808     {
809       \proxy_create:nnN { rawobjects }{ fwl ! #1 } \c_object_private_str
810       \proxy_push_member
811         {
812           \object_address:nn { rawobjects }{ fwl ! #1 }
813         }
814         { next }{ str }
815       }
816     }
817
818 \cs_new_protected:Nn \rawobjects_fwl_newnode:nnnNN
819 {
820   \rawobjects_fwl_inst:n { #1 }
821   \object_create:nnnNN
822     {
823       \object_address:nn { rawobjects }{ fwl ! #1 }
824     }
825     { #2 }{ #3 } #4 #5
826   }
827
828 </package>

```