The lt3rawobjects package

Paolo De Donato

Released on XXX Version 2.0

Contents

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 specification (`https://semver.org/`). In particular any major version update (for example from `1.2` to `2.0`) may introduce imcompatible 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 Instead changes introduced in minor and patch version updates are always backward compatible, and any withdrawn function is declared deprecated instead of being removed.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. Also in lt3rawobjects objects are collections of variables, called member variables, which can be retrieved from a string representing that object. Such string 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 automatically generated addresses.

Also control sequences have an address, but for them 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 C each object/structure has a *type* that tells the compiler how each object should be organized and instantiated in the memory. So if you need to create objects with the same structure you should first create a new `struct` entity and then create object with such type.

In lt3rawobjects objects are created from an existing object with a particular structure that holds all the needed informations to organize their variables. Such objects that can be used to instantiate new objects are calles *proxies* and the proxy object used to instantiate an object is its *generator*. In order to create new objects with a specified proxy you can use the `\object_create` functions.

Since proxies are themself 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`.

Once you've created you proxy object you should specify its member variables that will be created in each object initialized with such proxy. You can add a variable specification with the `\proxy_push_member` function. Once you've added all yor variables specifications you can use your proxy to create objects. You should never modify a proxy once you've used it to create at least one object, since these modifications won't be updated on already created objects, leading to hidden errors in subsequential code.

When you create a new variable specification with the `\proxy_push_member` you can notice the presence of ⟨*type*⟩ parameter. It represents the type of such variable and can be a standard type (like `tl`, `str`, `int`, `seq`, ...) or user defined types if the following functions are defined:

`\`⟨*type*⟩`_new:N` and `c` variant;

`\⟨type⟩_set_eq:NN` and `cN`, `Nc`, `cc` variants.

Every object, and so proxies too, is characterized by the following parameters:

- the *module* in which it has been created;
- the address of the proxy generator;
- a parameter saying if the object is *local* or *global*;
- a parameter saying if the object is *public* or *private*;
- zero or more member variables.

In a local/global/public/private object every member variable is declared local/global/public/private. Address of a member variable can be obtained with the `\object_member_adr` function, and you can instantiate new members that haven't been specified in its generator with the function `\object_new_member`. members created in this way aren't described by generator proxy, so its type can't be deduced and should be always specified in functions like `\object_member_adr` or `\object_member_use`.

4 Constants

This feature is available only from version 1.1 of lt3rawobjects. There're two different kinds of constants you can define on a object:

1. *near constants* are constants defined directly inside the associated object;
2. *remote constants* are constants that are defined instead on the generator proxy and so every object generated with that proxy can access the constant.

Currently it's possible to define only public constants, if you need private constants use member variables instead.

Notice that all near constants declared on a proxy are automatically remote constants for every generated object, but remote constants for a proxy aren't directly accessible by generated objects.

You can retrieve the address of a near constant with the `\object_nconst_adr` function and of a remote constant with `\object_rconst_adr`.

5 Methods (from version `2.0`)

Starting from version `2.0` you can define *methods* inside an object. There're two different types of methods:

- *constant methods* are methods created with `\cs_new:Nn` functions and can't be modified once thay're instantiated. Like constant members they can be near or remote;
- *variable methods* are instead created with `\cs_(g)set:Nn` functions and can be redefined after they're created.

comparing with C language constant methods are similar to C normal functions whereas variable ones to function pointers.

6 Library functions

6.1 Base object functions

\object_address:nn ⋆

\object_address:nn {⟨*module*⟩} {⟨*id*⟩}

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

\object_address_set:Nnn
\object_address_gset:Nnn

\object_address_set:nn ⟨*str var*⟩ {⟨*module*⟩} {⟨*id*⟩}

Stores the adress of selected object inside the string variable ⟨*str var*⟩.

From: 1.1

\object_if_exist_p:n ⋆
\object_if_exist_p:V ⋆
\object_if_exist:nTF ⋆
\object_if_exist:VTF ⋆

\object_if_exist_p:n {⟨*address*⟩}
\object_if_exist:nTF {⟨*address*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if an object was instantiated at the specified address.

From: 1.0

\object_get_module:n ⋆
\object_get_module:V ⋆
\object_get_proxy_adr:n ⋆
\object_get_proxy_adr:V ⋆

\object_get_module:n {⟨*address*⟩}
\object_get_proxy_adr:n {⟨*address*⟩}

Get the object module and its generator.

From: 1.0

\object_if_local_p:n ⋆
\object_if_local_p:V ⋆
\object_if_local:nTF ⋆
\object_if_local:VTF ⋆
\object_if_global_p:n ⋆
\object_if_global_p:V ⋆
\object_if_global:nTF ⋆
\object_if_global:VTF ⋆

\object_if_local_p:n {⟨*address*⟩}
\object_if_local:nTF {⟨*address*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the object is local or global.

From: 1.0

\object_if_public_p:n ⋆
\object_if_public_p:V ⋆
\object_if_public:nTF ⋆
\object_if_public:VTF ⋆
\object_if_private_p:n ⋆
\object_if_private_p:V ⋆
\object_if_private:nTF ⋆
\object_if_private:VTF ⋆

\object_if_public_p:n {⟨*address*⟩}
\object_if_public:nTF {⟨*address*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the object is public or private.

From: 1.0

6.2 Operating with member variables and constants

```
\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_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

```
\object_member_use:nnn       ⋆  \object_member_use:nnn {⟨address⟩} {⟨member name⟩} {⟨member type⟩}
\object_member_use:(Vnn|nnv) ⋆  \object_member_use:nn {⟨address⟩} {⟨member name⟩}
\object_member_use:nn        ⋆
\object_member_use:Vn        ⋆
```

Uses the specified member variable.

From: 1.0

```
\object_member_set_eq:nnnN                 ⋆  \object_member_set_eq:nnnN {⟨address⟩} {⟨member name⟩}
\object_member_set_eq:(nnvN|VnnN|nnnc|Vnnc) ⋆  {⟨member type⟩} ⟨variable⟩
\object_member_set_eq:nnN                  ⋆  \object_member_set_eq:nnN {⟨address⟩} {⟨member name⟩}
\object_member_set_eq:(VnN|nnc|Vnc)        ⋆  ⟨variable⟩
```

Sets the value of specified member equal to the value of ⟨*variable*⟩.

From: 1.0

```
\object_nconst_adr:nnn       ⋆  \object_nconst_adr:nnn {⟨address⟩} {⟨member name⟩} {⟨member type⟩}
\object_nconst_adr:(Vnn|vnn) ⋆
\object_rconst_adr:nnn       ⋆
\object_rconst_adr:Vnn       ⋆
```

Fully expands to the address of specified near/remote constant.

From: 1.1

```
\object_nconst_use:nnn ⋆  \object_nconst_use:nnn {⟨address⟩} {⟨member name⟩} {⟨member type⟩}
\object_nconst_use:Vnn ⋆
\object_rconst_use:nnn ⋆  Uses the specified near/remote constant.
\object_rconst_use:Vnn ⋆
```

From: 1.1

6.3 Methods

\object_ncmethod_adr:nnn ⋆
\object_ncmethod_adr:(Vnn|vnn) ⋆
\object_rcmethod_adr:nnn ⋆
\object_rcmethod_adr:Vnn ⋆
\object_vmethod_adr:nnn ⋆
\object_vmethod_adr:Vnn ⋆

\object_method_adr:nnn {⟨*address*⟩} {⟨*method name*⟩} {⟨*method variant*⟩}

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;
- variable method if \object_vmethod_adr is used;

From: 2.0

\object_new_cmethod:nnnn
\object_new_cmethod:Vnnn
\object_new_vmethod:nnnn
\object_new_vmethod:Vnnn

\object_new_vmethod:nnnn {⟨*address*⟩} {⟨*method name*⟩} {⟨*method arguments*⟩} {⟨*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 (for \object_new_cmethod) or to \cs_(g)set:Nn (for \object_new_vmethod).

From: 2.0

\object_cmethod_var:nnnn
\object_cmethod_var:Vnnn
\object_vmethod_var:nnnn
\object_vmethod_var:Vnnn

\object_cmethod_var:nnnn {⟨*address*⟩} {⟨*method name*⟩} {⟨*method arguments*⟩} {⟨*method variant*⟩}

Creates a new variant for the specified method.

From: 2.0

\object_ncmethod_call:nnn ⋆
\object_ncmethod_call:Vnn ⋆
\object_rcmethod_call:nnn ⋆
\object_rcmethod_call:Vnn ⋆
\object_vmethod_call:nnn ⋆
\object_vmethod_call:Vnn ⋆

\object_ncmethod_call:nnn {⟨*address*⟩} {⟨*method name*⟩} {⟨*method variant*⟩}

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

From: 2.0

6.4 Constant creation

Unlike normal variables, constants in LaTeX3 are created in different ways depending on the specified type. So we dedicate a new section only to collect some of these fuinctions readapted for near constants (remote constants are simply near constants created on the generator proxy).

\object_newconst_tl:nnn
\object_newconst_tl:Vnn
\object_newconst_str:nnn
\object_newconst_str:Vnn
\object_newconst_int:nnn
\object_newconst_int:Vnn
\object_newconst_clist:nnn
\object_newconst_clist:Vnn
\object_newconst_dim:nnn
\object_newconst_dim:Vnn
\object_newconst_skip:nnn
\object_newconst_skip:Vnn
\object_newconst_fp:nnn
\object_newconst_fp:Vnn

\object_newconst_⟨*type*⟩:nnn {⟨*address*⟩} {⟨*constant name*⟩} {⟨*value*⟩}

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

From: 1.1

\object_newconst_seq_from_clist:nnn
\object_newconst_seq_from_clist:Vnn

\object_newconst_seq_from_clist:nnn {⟨*address*⟩} {⟨*constant name*⟩} {⟨*comma-list*⟩}

Creates a `seq` constant which is set to contain all the items in ⟨*comma-list*⟩.

From: 1.1

\object_newconst_prop_from_keyval:nnn
\object_newconst_prop_from_keyval:Vnn

\object_newconst_prop_from_keyval:nnn {⟨*address*⟩} {⟨*constant name*⟩}
{
⟨*key*⟩ = ⟨*value*⟩, ...
}

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

From: 1.1

6.5 Proxy utilities and object creation

\object_if_proxy_p:n ⋆
\object_if_proxy_p:V ⋆
\object_if_proxy:nTF ⋆
\object_if_proxy:VTF ⋆

\object_if_proxy_p:n {⟨*address*⟩}
\object_if_proxy:nTF {⟨*address*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Test if the specified object is a proxy object.

From: 1.0

\object_test_proxy_p:nn ⋆
\object_test_proxy_p:Vn ⋆
\object_test_proxy:nnTF ⋆
\object_test_proxy:VnTF ⋆

\object_test_proxy_p:nn {⟨*object address*⟩} {⟨*proxy address*⟩}
\object_test_proxy:nnTF {⟨*object address*⟩} {⟨*proxy address*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Test if the specified object is generated by the selected proxy, where ⟨*proxy variable*⟩ 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 LuaLaTeX before 2019) it'll require slow processing. Don't use it in speed critical parts, instead use `\object_test_proxy:nN`.

From: 2.0

\object_test_proxy_p:nN ⋆
\object_test_proxy_p:VN ⋆
\object_test_proxy:nNTF ⋆
\object_test_proxy:VNTF ⋆

\object_test_proxy_p:nN {⟨*object address*⟩} ⟨*proxy variable*⟩
\object_test_proxy:nNTF {⟨*object address*⟩} ⟨*proxy variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Test if the specified object is generated by the selected proxy, where ⟨*proxy variable*⟩ is a string variable holding the proxy address. The :nN variant don't use e expansion, instead of :nn command, so it can be safetly used with older compilers.

From: 2.0

\c_proxy_address_str

The address of the proxy object in the rawobjects module.

From: 1.0

\object_create:nnnNN
\object_create:VnnNN

\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

\c_object_local_str
\c_object_global_str

Possible values for ⟨*scope*⟩ parameter.

From: 1.0

\c_object_public_str
\c_object_private_str

Possible values for ⟨*visibility*⟩ parameter.

From: 1.0

\object_create_set:NnnnNN
\object_create_set:NVnnNN
\object_create_gset:NnnnNN
\object_create_gset:NVnnNN

\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

\object_allocate_incr:NNnnNN
\object_allocate_incr:NNVnNN
\object_gallocate_incr:NNnnNN
\object_gallocate_incr:NNVnNN
\object_allocate_gincr:NNnnNN
\object_allocate_gincr:NNVnNN
\object_gallocate_gincr:NNnnNN
\object_gallocate_gincr:NNVnNN

\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

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

\proxy_create:nnN {⟨*module*⟩} {⟨*id*⟩} ⟨*visibility*⟩
\proxy_create_set:NnnN ⟨*str var*⟩ {⟨*module*⟩} {⟨*id*⟩} ⟨*visibility*⟩

Creates a global proxy object.

From: 1.0

\proxy_push_member:nnn
\proxy_push_member:Vnn

\proxy_push_member:nnn {⟨*proxy address*⟩} {⟨ *member name* ⟩} {⟨ *member type* ⟩}

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

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

\object_assign:nn {⟨*to address*⟩} {⟨*from address*⟩}

Assigns the content of each variable of object at ⟨*from address*⟩ to each correspective 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

7 Examples

Example 1

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

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

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

8 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 possiple 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, ...

9 Implementation

```
⟨*package⟩
⟨@@=rawobjects⟩
```

\c_object_local_str
\c_object_global_str
\c_object_public_str
\c_object_private_str

```
\str_const:Nn \c_object_local_str {loc}
\str_const:Nn \c_object_global_str {glo}
\str_const:Nn \c_object_public_str {pub}
\str_const:Nn \c_object_private_str {pri}

\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

```
\cs_new:Nn \object_address:nn {
  \tl_to_str:n { #1  _  #2 }
}
```

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

\object_address_set:Nnn
\object_address_gset:Nnn

Saves the address of an object into a string variable

```

\cs_new_protected:Nn \object_address_set:Nnn {
  \str_set:Nn #1 { #2  _  #3 }
}

\cs_new_protected:Nn \object_address_gset:Nnn {
  \str_gset:Nn #1 { #2  _  #3 }
}

```

(*End definition for* `\object_address_set:Nnn` *and* `\object_address_gset:Nnn`*. These functions are documented on page 4.*)

```
\cs_new:Nn \__rawobjects_object_modvar:n{
  c __ #1 _ MODULE _ str
}

\cs_new:Nn \__rawobjects_object_pxyvar:n{
  c __ #1 _ PROXY _ str
}

\cs_new:Nn \__rawobjects_object_scovar:n{
  c __ #1 _ SCOPE _ str
}

\cs_new:Nn \__rawobjects_object_visvar:n{
  c __ #1 _ VISIB _ str
}

\cs_generate_variant:Nn \__rawobjects_object_modvar:n { V }
\cs_generate_variant:Nn \__rawobjects_object_pxyvar:n { V }
\cs_generate_variant:Nn \__rawobjects_object_scovar:n { V }
\cs_generate_variant:Nn \__rawobjects_object_visvar:n { V }
```

\object_if_exist_p:n
\object_if_exist:nTF

Tests if object exists.

```
\prg_new_conditional:Nnn \object_if_exist:n { p, T, F, TF }
  {
    \cs_if_exist:cTF
      {
        \__rawobjects_object_modvar:n { #1 }
      }
      {
        \prg_return_true:
      }
      {
        \prg_return_false:
      }
  }

\prg_generate_conditional_variant:Nnn \object_if_exist:n { V }
  { p, T, F, TF }
```

(*End definition for* `\object_if_exist:nTF`*. This function is documented on page* 4.)

\object_get_module:n
\object_get_proxy_adr:n

Retrieve the name, module and generating proxy of an object

```
\cs_new:Nn \object_get_module:n {
  \str_use:c { \__rawobjects_object_modvar:n { #1 } }
}
\cs_new:Nn \object_get_proxy_adr:n {
  \str_use:c { \__rawobjects_object_pxyvar:n { #1 } }
}

\cs_generate_variant:Nn \object_get_module:n { V }
\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* 4.)

\object_if_local_p:n
\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

Test the specified parameters.

```
\prg_new_conditional:Nnn \object_if_local:n {p, T, F, TF}
{
  \str_if_eq:cNTF { \__rawobjects_object_scovar:n {#1} }
    \c_object_local_str
    {
      \prg_return_true:
    }
    {
      \prg_return_false:
    }
}

\prg_new_conditional:Nnn \object_if_global:n {p, T, F, TF}
{
  \str_if_eq:cNTF { \__rawobjects_object_scovar:n {#1} } \c_object_global_str
  {
    \prg_return_true:
```

```
  }
  {
    \prg_return_false:
  }
}

\prg_new_conditional:Nnn \object_if_public:n {p, T, F, TF}
{
  \str_if_eq:cNTF { \__rawobjects_object_visvar:n { #1 } } \c_object_public_str
  {
    \prg_return_true:
  }
  {
    \prg_return_false:
  }
}

\prg_new_conditional:Nnn \object_if_private:n {p, T, F, TF}
{
  \str_if_eq:cNTF { \__rawobjects_object_visvar:n {#1} } \c_object_private_str
  {
    \prg_return_true:
  }
  {
    \prg_return_false:
  }
}

\prg_generate_conditional_variant:Nnn \object_if_local:n { V }
  { p, T, F, TF }
\prg_generate_conditional_variant:Nnn \object_if_global:n { V }
  { p, T, F, TF }
\prg_generate_conditional_variant:Nnn \object_if_public:n { V }
  { p, T, F, TF }
\prg_generate_conditional_variant:Nnn \object_if_private:n { V }
  { p, T, F, TF }
```

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

`\object_member_adr:nnn`
`\object_member_adr:nn`

Get the address of a member variable

```

\cs_new:Nn \__rawobjects_scope:n
  {
    \object_if_global:nTF { #1 }
      {
        g
      }
      {
        \str_if_eq:cNTF { \__rawobjects_object_scovar:n { #1 } }
          \c__rawobjects_const_str
          {
            c
          }
          {
```

```
            l
          }
      }
  }

\cs_new:Nn \object_member_adr:nnn
  {
    \__rawobjects_scope:n { #1 }
    \object_if_private:nTF { #1 }
      {
        __
      }
      {
        _
      }
    #1 \tl_to_str:n { _ MEMBER _ #2 _ #3 }
  }

\cs_generate_variant:Nn \object_member_adr:nnn { Vnn, vnn, nnv }

\cs_new:Nn \object_member_adr:nn
  {
    \object_member_adr:nnv { #1 }{ #2 }
      {
        \object_rconst_adr:nnn { #1 }
          { #2 _ type }{ str }
      }
  }

\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 5.*)

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

```

\cs_new:Nn \object_member_type:nn
  {
    \object_rconst_use:nnn { #1 }
      { #2 _ type }{ str }
  }

```

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

```

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

\cs_new_protected:Nn \__rawobjects_force_scope:n
  {
    \bool_if:nF
```

```
      {
        \object_if_local_p:n { #1 } || \object_if_global_p:n { #1 }
      }
      {
        \msg_error:nnx { rawobjects }{ scoperr }{ #1 }
      }
  }

```

`\object_new_member:nnn` Creates a new member variable

```

\cs_new_protected:Nn \object_new_member:nnn
  {
    \__rawobjects_force_scope:n { #1 }
    \cs_if_exist_use:cT { #3 _ new:c }
      {
        { \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
      }
  }

\cs_generate_variant:Nn \object_new_member:nnn { Vnn, nnv }

```

(*End definition for* `\object_new_member:nnn`*. This function is documented on page 5.*)

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

```

\cs_new:Nn \object_member_use:nnn
  {
    \cs_if_exist_use:cT { #3 _ use:c }
      {
        { \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
      }
  }

\cs_new:Nn \object_member_use:nn
  {
    \object_member_use:nnv { #1 }{ #2 }
      {
        \object_rconst_adr:nnn { #1 }
          { #2 _ type }{ str }
      }
  }

\cs_generate_variant:Nn \object_member_use:nnn { Vnn, vnn, nnv }
\cs_generate_variant:Nn \object_member_use:nn { Vn }

```

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

`\object_member_set_eq:nnnN` `\object_member_set_eq:nnN` Set the value of a variable to a member.

```

\cs_new_protected:Nn \object_member_set_eq:nnnN
```

```
  {
    \__rawobjects_force_scope:n { #1 }
    \cs_if_exist_use:cT
      {
        #3 _ \object_if_global:nT { #1 }{ g } set _ eq:cN
      }
      {
        { \object_member_adr:nnn { #1 }{ #2 }{ #3 } } #4
      }
  }

\cs_generate_variant:Nn \object_member_set_eq:nnnN { VnnN, nnnc, Vnnc, nnvN }

\cs_new_protected:Nn \object_member_set_eq:nnN
  {
    \object_member_set_eq:nnvN { #1 }{ #2 }
      {
        \object_rconst_adr:nnn { #1 }
          { #2 _ type }{ str }
      } #3
  }

\cs_generate_variant:Nn \object_member_set_eq:nnN { VnN, nnc, Vnc }
```

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

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

```
\cs_new:Nn \object_nconst_adr:nnn
  {
    c _ #1 \tl_to_str:n { _ CONST _ #2 _ #3 }
  }

\cs_generate_variant:Nn \object_nconst_adr:nnn { Vnn, vnn }

\cs_new:Nn \object_rconst_adr:nnn
  {
    \object_nconst_adr:vnn { \__rawobjects_object_pxyvar:n { #1 } }
      { #2 }{ #3 }
  }

\cs_generate_variant:Nn \object_rconst_adr:nnn { Vnn }
```

(*End definition for* `\object_nconst_adr:nnn` *and* `\object_rconst_adr:nnn`*. These functions are documented on page 5.*)

`\object_nconst_use:nnn` `\object_rconst_use:nnn` Uses a near/remote constant.

```
\cs_new:Nn \object_nconst_use:nnn
  {
    \cs_if_exist_use:cT { #3 _ use:c }
      {
        { \object_nconst_adr:nnn { #1 }{ #2 }{ #3 } }
```

```
      }
  }

\cs_new:Nn \object_rconst_use:nnn
  {
    \cs_if_exist_use:cT { #3 _ use:c }
      {
        { \object_rconst_adr:nnn { #1 }{ #2 }{ #3 } }
      }
  }

\cs_generate_variant:Nn \object_nconst_use:nnn { Vnn }
\cs_generate_variant:Nn \object_rconst_use:nnn { Vnn }

```

(*End definition for* `\object_nconst_use:nnn` *and* `\object_rconst_use:nnn`. *These functions are documented on page* 5.)

`\object_newconst_tl:nnn`
`\object_newconst_str:nnn`
`\object_newconst_int:nnn`
`\object_newconst_clist:nnn`
`\object_newconst_dim:nnn`
`\object_newconst_skip:nnn`
`\object_newconst_fp:nnn`

Create constants

```

\cs_new_protected:Nn \__rawobjects_const_create:nnnn
  {
    \use:c { #1 _ const:cn }
      {
        \object_nconst_adr:nnn { #2 }{ #3 }{ #1 }
      }
      { #4 }
  }

\cs_new_protected:Nn \object_newconst_tl:nnn
  {
    \__rawobjects_const_create:nnnn { tl }{ #1 }{ #2 }{ #3 }
  }
\cs_new_protected:Nn \object_newconst_str:nnn
  {
    \__rawobjects_const_create:nnnn { str }{ #1 }{ #2 }{ #3 }
  }
\cs_new_protected:Nn \object_newconst_int:nnn
  {
    \__rawobjects_const_create:nnnn { int }{ #1 }{ #2 }{ #3 }
  }
\cs_new_protected:Nn \object_newconst_clist:nnn
  {
    \__rawobjects_const_create:nnnn { clist }{ #1 }{ #2 }{ #3 }
  }
\cs_new_protected:Nn \object_newconst_dim:nnn
  {
    \__rawobjects_const_create:nnnn { dim }{ #1 }{ #2 }{ #3 }
  }
\cs_new_protected:Nn \object_newconst_skip:nnn
  {
    \__rawobjects_const_create:nnnn { skip }{ #1 }{ #2 }{ #3 }
  }
\cs_new_protected:Nn \object_newconst_fp:nnn
```

```
  {
    \__rawobjects_const_create:nnnn { fp }{ #1 }{ #2 }{ #3 }
  }

\cs_generate_variant:Nn \object_newconst_tl:nnn { Vnn }
\cs_generate_variant:Nn \object_newconst_str:nnn { Vnn }
\cs_generate_variant:Nn \object_newconst_int:nnn { Vnn }
\cs_generate_variant:Nn \object_newconst_clist:nnn { Vnn }
\cs_generate_variant:Nn \object_newconst_dim:nnn { Vnn }
\cs_generate_variant:Nn \object_newconst_skip:nnn { Vnn }
\cs_generate_variant:Nn \object_newconst_fp:nnn { Vnn }
```

(*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.

```
\cs_new_protected:Nn \object_newconst_seq_from_clist:nnn
  {
    \seq_const_from_clist:cn
      {
        \object_nconst_adr:nnn { #1 }{ #2 }{ seq }
      }
      { #3 }
  }

\cs_generate_variant:Nn \object_newconst_seq_from_clist:nnn { Vnn }
```

(*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.

```
\cs_new_protected:Nn \object_newconst_prop_from_keyval:nnn
  {
    \prop_const_from_keyval:cn
      {
        \object_nconst_adr:nnn { #1 }{ #2 }{ prop }
      }
      { #3 }
  }

\cs_generate_variant:Nn \object_newconst_prop_from_keyval:nnn { Vnn }
```

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

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

```
\cs_new:Nn \object_ncmethod_adr:nnn
  {
    #1 \tl_to_str:n { _ CMETHOD _ #2 : #3 }
  }

\cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
```

```
\cs_new:Nn \object_rcmethod_adr:nnn
  {
    \object_ncmethod_adr:vnn
      {
        \__rawobjects_object_pxyvar:n { #1 }
      }
      { #2 }{ #3 }
  }

\cs_new:Nn \object_vmethod_adr:nnn
  {
    \object_if_global:nTF { #1 }
      {
        #1 \tl_to_str:n { _ GMETHOD _ #2 : #3 }
      }
      {
        #1 \tl_to_str:n { _ LMETHOD _ #2 : #3 }
      }
  }

\cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
\cs_generate_variant:Nn \object_rcmethod_adr:nnn { Vnn }
\cs_generate_variant:Nn \object_vmethod_adr:nnn { Vnn }
```

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

`\object_new_cmethod:nnnn`
`\object_new_vmethod:nnnn`

Creates a new method

```
\cs_new_protected:Nn \object_new_cmethod:nnnn
  {
    \cs_new:cn
  {
    \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
  }
  { #4 }
  }

\cs_new_protected:Nn \object_new_vmethod:nnnn
  {
    \object_if_global:nTF { #1 }
      {
        \cs_gset:cn
          {
            \object_vmethod_adr:nnn
              { #1 }{ #2 }{ #3 }
          }
          { #4 }
      }
      {
        \cs_set:cn
          {
```

```
            \object_vmethod_adr:nnn
              { #1 }{ #2 }{ #3 }
          }
          { #4 }
      }
  }

\cs_new_protected:Nn \object_new_method_protected:nnnn
  {
    \cs_new_protected:cn
  {
    \object_method_adr:nnn { #1 }{ #2 }{ #3 }
  }
  { #4 }
  }

\cs_new_protected:Nn \object_new_method_nopar:nnnn
  {
    \cs_new_nopar:cn
  {
    \object_method_adr:nnn { #1 }{ #2 }{ #3 }
  }
  { #4 }
  }

\cs_new_protected:Nn \object_new_method_protected_nopar:nnnn
  {
    \cs_new_protected_nopar:cn
  {
    \object_method_adr:nnn { #1 }{ #2 }{ #3 }
  }
  { #4 }
  }

\cs_generate_variant:Nn \object_new_cmethod:nnnn { Vnnn }
\cs_generate_variant:Nn \object_new_vmethod:nnnn { Vnnn }

\cs_generate_variant:Nn \object_new_method_protected:nnnn { Vnnn }
\cs_generate_variant:Nn \object_new_method_nopar:nnnn { Vnnn }
\cs_generate_variant:Nn \object_new_method_protected_nopar:nnnn { Vnnn }

```

(*End definition for* `\object_new_cmethod:nnnn` *and* `\object_new_vmethod:nnnn`. *These functions are documented on page* 6.)

`\object_cmethod_var:nnnn`
`\object_vmethod_var:nnnn`

Generates a method variant.

```

\cs_new_protected:Nn \object_cmethod_var:nnnn
  {
    \cs_generate_variant:cn
  {
    \object_cmethod_adr:nnn { #1 }{ #2 }{ #3 }
  }
  { #4 }
```

```
  }

\cs_new_protected:Nn \object_vmethod_var:nnnn
  {
    \cs_generate_variant:cn
  {
    \object_vmethod_adr:nnn { #1 }{ #2 }{ #3 }
  }
  { #4 }
  }

\cs_generate_variant:Nn \object_cmethod_var:nnnn { Vnnn }
\cs_generate_variant:Nn \object_vmethod_var:nnnn { Vnnn }
```

(*End definition for* `\object_cmethod_var:nnnn` *and* `\object_vmethod_var:nnnn`*. These functions are documented on page 6.*)

`\object_ncmethod_call:nnn`
`\object_rcmethod_call:nnn`
`\object_vmethod_call:nnn`

Calls the specified method.

```
\cs_new:Nn \object_ncmethod_call:nnn
  {
    \use:c
  {
    \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
  }
  }

\cs_new:Nn \object_rcmethod_call:nnn
  {
    \use:c
  {
    \object_rcmethod_adr:nnn { #1 }{ #2 }{ #3 }
  }
  }

\cs_new:Nn \object_vmethod_call:nnn
  {
    \use:c
  {
    \object_vmethod_adr:nnn { #1 }{ #2 }{ #3 }
  }
  }

\cs_generate_variant:Nn \object_ncmethod_call:nnn { Vnn }
\cs_generate_variant:Nn \object_rcmethod_call:nnn { Vnn }
\cs_generate_variant:Nn \object_vmethod_call:nnn { Vnn }
```

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

`\c_proxy_address_str`

The address of the `proxy` object.

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

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

Source of `proxy` object

```
\str_const:cn { \__rawobjects_object_modvar:V \c_proxy_address_str }
  { rawobjects }
\str_const:cV { \__rawobjects_object_pxyvar:V \c_proxy_address_str }
  \c_proxy_address_str
\str_const:cV { \__rawobjects_object_scovar:V \c_proxy_address_str }
  \c__rawobjects_const_str
\str_const:cV { \__rawobjects_object_visvar:V \c_proxy_address_str }
  \c_object_public_str

\seq_const_from_clist:cn
  {
    \object_member_adr:Vnn \c_proxy_address_str { varlist }{ seq }
  }
  { varlist }

\object_newconst_str:Vnn \c_proxy_address_str { varlist_type }{ seq }

```

`\object_if_proxy_p:n`
`\object_if_proxy:nTF`

Test if an object is a proxy.

```
\prg_new_conditional:Nnn \object_if_proxy:n {p, T, F, TF}
  {
    \object_test_proxy:nNTF { #1 }
  \c_proxy_address_str
      {
        \prg_return_true:
      }
      {
        \prg_return_false:
      }
  }

```

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

`\object_test_proxy_p:nn`
`\object_test_proxy:nnTF`
`\object_test_proxy_p:nN`
`\object_test_proxy:nNTF`

Test if an object is generated from selected proxy.

```
\prg_generate_conditional_variant:Nnn \str_if_eq:nn { ve }{ TF }

\prg_new_conditional:Nnn \object_test_proxy:nn {p, T, F, TF}
  {
    \str_if_eq:veTF { \__rawobjects_object_pxyvar:n { #1 } }
  { #2 }
      {
        \prg_return_true:
      }
      {
        \prg_return_false:
      }
  }

\prg_new_conditional:Nnn \object_test_proxy:nN {p, T, F, TF}
```

```
  {
    \str_if_eq:cNTF { \__rawobjects_object_pxyvar:n { #1 } }
  #2
      {
        \prg_return_true:
      }
      {
        \prg_return_false:
      }
  }

\prg_generate_conditional_variant:Nnn \object_test_proxy:nn { Vn }{p, T, F, TF}
\prg_generate_conditional_variant:Nnn \object_test_proxy:nN { VN }{p, T, F, TF}

```

(*End definition for* `\object_test_proxy:nnTF` *and* `\object_test_proxy:nNTF`. *These functions are documented on page* 7.)

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

Creates an object from a proxy

```

\msg_new:nnn { aa }{ mess }{ #1 }

\msg_new:nnnn { rawobjects }{ notproxy }{ Fake ~ proxy }
  {
    Object ~ #1 ~ is ~ not ~ a ~ proxy.
  }

\cs_new_protected:Nn \__rawobjects_force_proxy:n
  {
    \object_if_proxy:nF { #1 }
      {
        \msg_error:nnn { rawobjects }{ notproxy }{ #1 }
      }
  }

\cs_new_protected:Nn \__rawobjects_create_anon:nnnNN
  {

    \__rawobjects_force_proxy:n { #1 }

    \str_const:cn { \__rawobjects_object_modvar:n { #2 } }{ #3 }
    \str_const:cx { \__rawobjects_object_pxyvar:n { #2 } }{ #1 }
    \str_const:cV { \__rawobjects_object_scovar:n { #2 } } #4
    \str_const:cV { \__rawobjects_object_visvar:n { #2 } } #5

    \seq_map_inline:cn
      {
        \object_member_adr:nnn { #1 }{ varlist }{ seq }
      }
      {
        \object_new_member:nnv { #2 }{ ##1 }
          {
            \object_nconst_adr:nnn { #1 }{ ##1 _ type }{ str }
          }
```

```
      }
  }

\cs_new_protected:Nn \object_create:nnnNN
  {
    \__rawobjects_create_anon:nnnNN { #1 }{ \object_address:nn { #2 }{ #3 } }
      { #2 } #4 #5
  }

\cs_new_protected:Nn \object_create_set:NnnnNN
  {
    \object_create:nnnNN { #2 }{ #3 }{ #4 } #5 #6
    \str_set:Nx #1 { \object_address:nn { #3 }{ #4 } }
  }

\cs_new_protected:Nn \object_create_gset:NnnnNN
  {
    \object_create:nnnNN { #2 }{ #3 }{ #4 } #5 #6
    \str_gset:Nx #1 { \object_address:nn { #3 }{ #4 } }
  }

\cs_generate_variant:Nn \object_create:nnnNN { VnnNN }
\cs_generate_variant:Nn \object_create_set:NnnnNN { NVnnNN }
\cs_generate_variant:Nn \object_create_gset:NnnnNN { NVnnNN }

```

(*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` `\object_gallocate_incr:NNnnNN` `\object_allocate_gincr:NNnnNN` `\object_gallocate_gincr:NNnnNN`

Create an address and use it to instantiate an object

```

\cs_new:Nn \__rawobjects_combine:nn
  {
    anon . #2 . #1
  }

\cs_generate_variant:Nn \__rawobjects_combine:nn { Vn }

\cs_new_protected:Nn \object_allocate_incr:NNnnNN
  {
    \object_create_set:NnnnNN #1 { #3 }{ #4 }
      {
        \__rawobjects_combine:Vn #2 { #3 }
      }
      #5 #6

      \int_incr:N #2
  }

\cs_new_protected:Nn \object_gallocate_incr:NNnnNN
  {
    \object_create_gset:NnnnNN #1 { #3 }{ #4 }
      {
        \__rawobjects_combine:Vn #2 { #3 }
```

```
      }
      #5 #6

      \int_incr:N #2
  }

\cs_generate_variant:Nn \object_allocate_incr:NNnnNN { NNVnNN }

\cs_generate_variant:Nn \object_gallocate_incr:NNnnNN { NNVnNN }

\cs_new_protected:Nn \object_allocate_gincr:NNnnNN
  {
    \object_create_set:NnnnNN #1 { #3 }{ #4 }
      {
        \__rawobjects_combine:Vn #2 { #3 }
      }
      #5 #6

      \int_gincr:N #2
  }

\cs_new_protected:Nn \object_gallocate_gincr:NNnnNN
  {
    \object_create_gset:NnnnNN #1 { #3 }{ #4 }
      {
        \__rawobjects_combine:Vn #2 { #3 }
      }
      #5 #6

      \int_gincr:N #2
  }

\cs_generate_variant:Nn \object_allocate_gincr:NNnnNN { NNVnNN }

\cs_generate_variant:Nn \object_gallocate_gincr:NNnnNN { NNVnNN }

```

(*End definition for* `\object_allocate_incr:NNnnNN` *and others. These functions are documented on page 8.*)

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

Creates a new proxy object

```
\cs_new_protected:Nn \proxy_create:nnN
  {
    \object_create:VnnNN \c_proxy_address_str { #1 }{ #2 }
      \c_object_global_str #3
  }

\cs_new_protected:Nn \proxy_create_set:NnnN
  {
    \object_create_set:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
      \c_object_global_str #4
  }

```

```
\cs_new_protected:Nn \proxy_create_gset:NnnN
  {
    \object_create_gset:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
      \c_object_global_str #4
  }

```

(*End definition for* `\proxy_create:nnN`, `\proxy_create_set:NnnN`, *and* `\proxy_create_gset:NnnN`. *These functions are documented on page 8.*)

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

```
\cs_new_protected:Nn \proxy_push_member:nnn
  {
    \__rawobjects_force_scope:n { #1 }
    \object_newconst_str:nnn { #1 }{ #2 _ type }{ #3 }
    \seq_gput_left:cn
      {
        \object_member_adr:nnn { #1 }{ varlist }{ seq }
      }
      { #2 }
  }

\cs_generate_variant:Nn \proxy_push_member:nnn { Vnn }

```

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

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

```
\cs_new_protected:Nn \object_assign:nn
  {
    \seq_map_inline:cn
      {
        \object_member_adr:vnn
          {
            \__rawobjects_object_pxyvar:n { #1 }
          }
          { varlist }{ seq }
      }
      {
        \object_member_set_eq:nnc { #1 }{ ##1 }
          {
            \object_member_adr:nn{ #2 }{ ##1 }
          }
      }
  }

\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

```

\cs_new_protected:Nn \rawobjects_fwl_inst:n
  {
    \object_if_exist:nF
      {
```

```
        \object_address:nn { rawobjects }{ fwl ! #1 }
      }
      {
        \proxy_create:nnN { rawobjects }{ fwl ! #1 } \c_object_private_str
        \proxy_push_member
          {
            \object_address:nn { rawobjects }{ fwl ! #1 }
          }
          { next }{ str }
      }
  }

\cs_new_protected:Nn \rawobjects_fwl_newnode:nnnNN
  {
    \rawobjects_fwl_inst:n { #1 }
    \object_create:nnnNN
      {
        \object_address:nn { rawobjects }{ fwl ! #1 }
      }
      { #2 }{ #3 } #4 #5
  }

⟨/package⟩
```