Klone
C Interface
Manual

V **1.0**
April 1994

Colas Nahaboo
Koala Project
BULL Research
`colas@sa.inria.fr`

Copyright

Author:	Colas NAHABOO -- BULL Research FRANCE -- Koala Project Wool Language and GWM X11 Window Manager Architect.
Internet:	colas@sophia.inria.fr
Surface Mail:	Colas NAHABOO INRIA B.P.93 - 06902 Sophia Antipolis Cedex FRANCE
Voice phone:	(33) 93.65.77.70
Fax:	(33) 93 65 77 66
Telex:	97 00 50 F

Acknowledgments

Using the standalone Klone interpreter 4
Writing Klone scripts 5
Linking your application with Klone 5
Calling Klone from C 5
Calling C from Klone 6
Defining new Klone primitive data types 7
Static definitions (.h) 7
Dynamic initialisations (.c) 7
Use from Klone 9
Taking advantage of Klone for your C programming 9
The garbage collector 9
The object model 9
The parsing and printing engine 9
Linking with Klone 10
Creating a new built-in Klone type 10
Conventions for creating a new type 12
Selectors declarations 12
Types 13
Type declarations 13
Functions on types 14
Traits 15
Variable declarations 16
Function declarations 16
Utility routines 16
The Garbage Collector 18
Generalities: What is a Garbage Collector? 18
Klone GC Principles 19
Implementation 20

CHAPTER 1 Tutorial

Here are some “cookbook recipes” on how to practically embed Klone into a C application. Exemples will gradually show more and more intricate use of Klone into another application. See the “C interface to Klone Reference” chapter. for complete reference.

1.1 Using the standalone Klone interpreter

Just type the command **klone**. You can then type expressions which are interpreted by Klone, which prints the retuned value. The Klone interpreter prompts you with the character **?** to invite your typing, and prints the result prefixed by **=**. If you end a line without closing the top-level expression, the input line is indented by the level of indentation, with a level represented by two spaces.

```
opossum$ klone                              Unix command
KLONE toplevel 1.0                          Banner
? (+ 2 2)
 = 4
? (- 10                                     A multi-line expression
?   (+ 2
?     2)
?    )
 = 6
?
```

It is recommended that you use some kind of front-end to this system to provide line editing, access to the history of typed lines, etc... For instance you may use it under the **Emacs** editor or via the pseud-tty handlers **fep** (Front End Processor) or **ile** (Interactive Line Editing), all being freely available.

```
opossum$ fep klone
```

Experimenting with the interpreted will be greatly eased if you enter the interactive Klone debugger each time an error happens. For this, just type **(kbd t)** in Klone, or put this line into your Klone profile **.klonerc** (in the current directory) that is read at the start of interactive sessions.

1.2 Writing Klone scripts

A very good way to get used to klone is to begin to write shell scripts in klone rather than in shells (or awk, perl, tcl, etc...). The klone interpreter supports the Unix `#!` convention, so by putting at the start of you script file the line:

```
#!/usr/local/bin/klone
```

(assumming this is the actual path of the installed executable of the klone interpreter on your system) you can write unix commands in Klone as easily as you would write shell scripts.

Emacs users may want to make the first line of scripts be rather the following to be in lisp-mode for editing Klone scripts:

```
#!/usr/local/bin/klone                              -*- lisp -*-
```

1.3 Linking your application with Klone

Suppose that you have an application and you want to link it with klone. What you may do is something along the lines:

```
... your include files...
#include "klone/KlLib.h"          /* includes all needeed klone files */

main () {
    /* ... your initilization ode ... */
    KlInit();                     /* initialize Klone interpreter */
    /* the rest of your application */
}
```

And then compile your program with something like:

```
cc -I/usr/local/include my_program.c -o my_program
    -L/usr/local/lib -lklone
```

Depending, of course of where the klone includes and library are located on your system. Here at INRIA, they are in /net/koala/include and /net/koala/lib/klone/sun4 respectively.

The previous exemple do not use the default Klone main loop. If you want to use the Klone main loop to provide a lips-like interpreter augmented with your custom routines, the declaration should be something like:

```
#include "klone/KlLib.h"       /* includes all needeed klone
files */

main () {
    /* ... your initilalization ode ... */
     KlInit();                      / * initialize Klone interpreter */
     for(;;)
          KlPoolToplevel();         /* read/eval/print */
}
```

1.4 Calling Klone from C

Calling Klone from C can be done in many ways. The simplest form is to give a C string (for instance `"(+ 2 2)"`) containing the text of a klone expression to be executed. This is accomplished by the following call:

```
KlExecuteStringNoReturn("(+ 2 2)");
```

A more elaborate form is to get the value returned by the expression:

```
KlO result = KlExecuteString(gets(buffer));
```

You can then look at the result, which is a Klone object, i.e. a pointer to a C structure. For instance:

```
int expected_number;
KlO result = KlExecuteString(gets(buffer));
if (result && KlIsANumber(result)) /* no error and valid number */
    expected_number = ((KlNumber) result)->number;
else
    expected_number = 0;
```

A word on the garbage colllector (GC) is now in order. The resulting value **`result`**, has been just created, and may thus be freed at the next GC invocation if no C object declares that it is interested in the result, Although Klone guarantees that **`result`** wont be freed as long as you stay in the scope of the current C function.

If you call Klone from the toplevel loop of your application (in a Motif Callback for instance) you will have to call the GC explicitely. This is done by a call to **`KlGC()`**. Use of the GC is simply done by declaring **`KlGCMark()`** before calling Klone code, and **`KlGC()`** afterwards. Here is the completed code of the **`read_klone_number`** function which reads a line from the standard input, parses it by Klone and return the resulting number, or **`0`** if it was not a number.

```
int
read_klone_number()
{
    int expected_number;
    KlGCMark();
    char buffer[1000];
    KlO result = KlExecuteString(gets(buffer)); /* call Klone */
    if (result && KlIsANumber(result))/* access Klone Object */
        expected_number = ((KlNumber) result)->number;
    else
        expected_number = 0;           /* an error occured */
    KlGC();
    return expected_number;
}
```

1.5 Calling C from Klone

To call your application C routines from klone, you must declare them as Klone functions. But these functions will be handled Klone objects and must return Klone objects, so you may have to write *wrapper* functions that will convert the parameters to C and the result back to Klone for most of your already-existing functions. By convention, wrapper function have the same name as the wrapped function, but with a **`Kl`** appended.

Here is an example: make the chmod Unix function available from Klone.

```
/* chmod signature, for info*/
/* int chmod(path, mode)
 * char *path;
 * mode_t mode;
 */

/* Wrapper function: chmodKl */
KlNumber
chmodKl(klpath, klmode)
    KlString klpath;
    KlNumber klmode;
{
    KlMustBeString(klpath, 0);         /* check validity of first arg */
    KlMustBeNumber(klmode, 1);         /* check validity of first arg */
    KlNumberMake(                    /* convert result to a Klone number */
        chmod(klpath->string, klmode->number));/* Call wrapped */
}

/* Now we declare this function to klone, under the klone name "chmod" */
KlDeclareSubr(chmodKl, "chmod", 2);/* 2 is the number of arguments */
```

1.6 Defining new Klone primitive data types

One of the main advantages of Klone is to be adaptable at run-time to use the data types relevant to the application. This avoids both the overhead of having to convert the application data back and forth to the interpreter native format, and enforce a better parameter checking mecanism. For instance, most lisp interpreters only deal with lists and vectors, thus if you want to embed one into a mailer, for instance, you will have to convert all your data types (mail, adress, header fields...) to strings, whereas in Klone you just export to Klone your already-existing C datatypes, and they become first-class data types used exactly like the primitive Klone data types, such as lists, hashtables, which are in fact also defined at run-time during Klone initialization.

Suppose in the following that we want to define a new type of object: a **stack**. These objects will have two fields, the number of elements in the stack (pointer to the top of the stack), and a pointer to the data (base of the stack).

1.6.1 Static definitions (.h)

First, you need to have a definition of the Klone objects of type stack, seen from C. Let's call it **`Stack`**. Its declaration will take the form:

```
typedef struct StackStruct {
    KlKLONE_HEADER;
    int top;                    /* number of elements */
    KlO *base;                  /* array of elements of any Klone type */
} *Stack;                       /* a Klone type is a pointer to a structure
*/
```

`KlO` being the generic type of all Klone objects.

Then you have to define a C variable to contain the Klone type corresponding to **`Stack`**s, a Klone object (KlO):

```
extern KlType StackType;
```

1.6.2 Dynamic initialisations (.c)

Then you must define the functions operating on this type (methods) in a .c file, and create a **`StackInit`** C function containing all the run-time initialisations that your code should call some time after initialising the Klone interpreter by a call to **`KlInit`**.

```
StackInit() {
    /* declare the Klone type */
    KlDeclareType(&StackType, "Stack",
        sizeof(struct StackStruct));
    /* traits declarations, if any ... */
    /* declare methods */
    KlDeclareMethod(StackType, KlSelFree, StackFree);
    KlDeclareMethod(StackType, KlSelLength, StackLength);
    KlDeclareMethod(StackType, KlSelPrint, StackPrint);
    /* declare functions specific to this type */
    KlDeclareSubr(StackPush, "stack-push", 2);
    KlDeclareSubr(StackPop, "stack-pop", 1);
    KlDeclareSubr(StackMake, "stack-make", NARY);
    /* create special objects if needed */
}
```

With the correponding C functions **`StackFree`**, **`StackLength`**, **`StackPrint`**, **`StackMake`**, **`StackPush`**, **`StackPop`** being defined as:

```
/* StackFree: called to deallocate a stack when no more used * /
KlO StackFree(stack)
    Stack stack;
{
    int top = stack->top;
    while (top--)           /* decrement refcount of objects pointed to */
        KlDecRef(stack->base[top]);
```

```
    KlFree(stack);           /* actually free the memory */
    return (KlO) stack;
}

/* StackLength: returns the sixe of a stack */
KlO StackLength(stack)
    Stack stack;
{
    return (KlO) KlNumberMake(stack->top);
}

/* StackPrint: prints a stack nicely, one element per line*/
KlO StackPrint(stack, stream)
    Stack stack;
    KlStream stream;
{
    int i;
    for (i = stack->top - 1; i > 0; i--) { /* print in reverse order */
        KlSPrint(stack->base[i], stream);
        KlSPutc('\n', stream);
    }
    return (KlO) stack;
}

/* StackMake: create a stack and put the optional arguments in it */
Stack StackMake(argc, argv)
    int argc;
    KlO *argv;               /* the list of arguments in traditional C format */
{
    int i;

    /* allocate the object */
    Stack stack = (Stack) KlOMake(StackType);
    /* initialize the fields */
    stack->top = 0;
    stack->base = (KlO *)Malloc(0);
    /* parse the variable-length list of arguments and pushes them on the stack */
    for (i=0; i < argc; i++)
        StackPush(stack, argv[i]);
    return stack;
}

/* StackPush: pushes an object on the stack */
KlO StackPush(stack, object)
    Stack stack;
    KlO object;
{
    /* check argument type */
    KlArgumentMustBe(stack, 0, StackType);
    /* extend list and push it */
    stack->top++;
    stack->base = (KlO *)
        Realloc(stack->base, stack->top * sizeof(KlO));
    stack->base[stack->top - 1] = object;
    /* increment reference count of pointed object */
    KlIncRef(object);
    return (KlO) stack;
}

/* StackPop: removes an elemnt from the top and returns it */
KlO StackPop(stack)
    Stack stack;
{
    /* check argument type */
    KlArgumentMustBe(stack, 0, StackType);
    if (stack->top) {
        KlO elt = stack->base[--(stack->top)];
        WlDecRef(elt);
        return elt;
    } else {
        return NIL;
    }
```

```
}
```

1.6.3 Use from Klone

Then, in the Klone process you have access to these new symbols:

`Stack` the type of stack objects

`stack-make`, the function to build a new stack:

```
(setq s (stack-make 'a 1 "foo"))
(typep s Stack) ⟹ Stack
s               ⟹ foo
                   1
                   a
```

`stack-pop` and **`stack-push`**:

```
(stack-pop s)   ⟹ foo
s               ⟹ 1
                   a
```

1.7 Taking advantage of Klone for your C programming

Klone can also be used for the primitive (but fast) object layer it provides to the C programmer. By declaring the C structures used as Klone types (via KlDeclareType), you gain:

Run-time typing: Any object can be identified by its type field, and complex inheritance schemes can be designed via the traits system.

Message sending: By the KlSend function.

Garbage Collection: By the simple reference count system provided on Klone objects.

Dynamic Structures: Klone built in lists and hash tables are valuable objects for the C programmer.

Parsing and printing: Klone can be used to store and retreive data, or communicating with other applications in a text form. Designing an application save file format as Klone expressions greatly simplifies parsing.

1.7.1 The garbage collector

1.7.2 The object model

1.7.3 The parsing and printing engine

CHAPTER 2

C Interface to Klone Reference

2.1 Linking with Klone

To use Klone, you must:

- Include the file **`"KlLib.h"`** into all C modules using Klone objects
- add a call to **`KlMallocInit()`** as the first line of your main loop if you use the version of **`libklone.a`** which redefines the standard malloc routines
- Call the routine **`KlInit`** with 3 arguments which are either NULL or pointer to C functions with no arguments you would have coded to perform:
  1) declaration of new selectors
  2) declaration of new types
  3) miscellaneous initialisations that must be performed after all built-in functions and objects are defined, but before reading the user profile

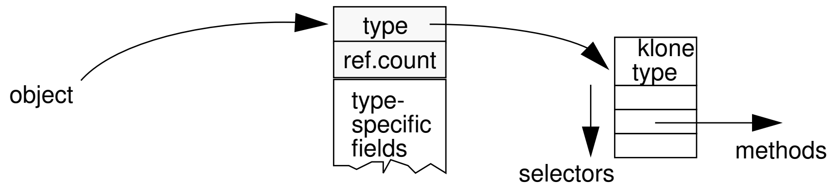
Then you must call regularily when coming back to your main loop the Klone function **`KlToplevelGC`** without arguments to call the garbage collector. Klone expressions can then be evaluated by calling **`KlExecuteString`** or **`KlExecuteStringNoReturn`** on the C string containing the text of the Klone expression, the first form returning the Klone object result of the evaluation of the expression, the second returning no argument..

2.2 Creating a new built-in Klone type

Klone is designed for easy extension of its type system. Being intended to be used as an embedded interpreter into a host application, the recommended practise is to provide access to the application-specific data types by making them visble to the Klone profile writer as built-in Klone types. This is opposed to the traditional use of embedded interpreters where the application needs to convert its internal data types to and from the interpreter natural types.

For instance, to embed Klone into a mailer application, having built-in types such as an internet adress, a mail message, the natural way would be to declare new primitive Klone types representing adresses and messages, these declarations being done at run-time during the initialization of the Klone library. A lot of interpreters would require the embedder to use the language strings to represent adresses and perhaps lists of strings to reprsent messages. The klone approach provides the safety of run-time typing of objects and the «abstract type» approch of hiding the details of the implementation to the programmer and the efficiency of not having the overhead of a systematic conversion of data between the application and its embedded interpreter.

All Klone objects are built around the same pattern:



A Klone object is a pointer to a C structure whose first two fields are common to all Klone objects: **`type`**, the type of the object, which is a Klone object of type **`KlType`** (and thus a pointer to a structure), and the **`reference_count`**, internally used to manage the count of objects pointing to it. Following this field are defined the fields pertaining specifically to this type.

As an example, the Integer (the simplest Klone object) and the HashTable (a more complex example) Klone types are defined by:

```
typedef struct _KlNumber {
    KlKLONE_HEADER;                         /* common header */
    int number;                             /* the only field */
} *KlNumber;

typedef struct _KlHashCell {                /* a cell of the linked lists */
    struct _KlHashCell *next;
    KlO key;
    KlO object;
} *KlHashCell;

typedef struct _KlHash {
    KlKLONE_HEADER;
    int size;                               /* number of stored objs */
    KlHashCell *table;                      /* array of linked lists */
    int limit;                              /* room for N entries */
    unsigned int ref_counted;               /* what is refcounted? */
} *KlHash;
```

The **`KlKLONE_HEADER`** macro is provided to define the common fields to all Klone types. Thus a definition for the type Foobar would look like:

```
typedef struct _Foobar {
    KlKLONE_HEADER;
    /* specific fields here */
} *Foobar;

Foobar foo;                         /* a declaration for a Foobar */
```

A Klone type is basically the standard Klone header (so that KlType objects are standalone Klone objects), followed by internal information, and by an array of methods for the objects of this type. Selectors for this type are thus just numbers that are used as indexes into the type viewed as an array of methods (pointers to C functions that take at least the object as first argument). All selector names are prefixed by **`KlSel`**. The list of predefined selectors is:

`KlSelEval`	evaluates the object
`KlSelPrint`	prints object on 2nd arg stream
`KlSelFree`	frees the object, called by the GC
`KlSelExecute`	called when evaluating a list with object as first elemnt. The whole call list is passed as the 2nd argument, and arguments must be evaluated if needed
`KlSelApply`	as `KlSelExecute`, but arguments are not evaluated
`KlSelEqual`	called on obejct and other object to test with
`KlSelCopy`	must return a copy of the object
`KlSelSetq`	assigns the object to the value given in second argument

`KlSelGetCValue`	the only function that must return a C object, not a klone object. used for automatically converting things in some C codes
`KlSelMake`	creates an instance of type out of a C obejct given as argument
`KlSelAdd`	returns the addition of rest of arguments to the first object, without physically modifying the object. Concatenates strings, add numbers...
`KlSelGet`	get a value of a slot named by the 2nd argument of the object. If slot is not found, should return the evaluation of the third argument
`KlSelPut`	sets the value of a slot named by the 2nd argument of the object. If slot is not found, should return an error KlE_NO_SLOT
`KlSelDelete`	physically removes a slot named by the 2nd argument from object
`KlSelInsert`	physically adds a slot named by the 2nd argument from object with value being the third argument
`KlSelNth`	should return the N-th element of object, N being the 2nd argument
`KlSelLength`	should return the length of object as a Klone number
`KlSelHash`	should return the hash function of the object as a C number

2.3 Conventions for creating a new type

New Klone types are defined in two files: a definition (.h) file and an implementation (or body) file (.c) of the same basename. These files consist of these sections, for a type Foobar:

Definition file:

```
C type declarations of type Foobar
Function prototypes
Macros (type-checking macros: KlIsFoobar...)
Declaration of the Klone type (KlType) C variable(s) Foobar-
Type
```

Implementation file:

```
#include "KlLib.h"
declarations of functions operating on instances of the type
FoobarInit function to initialize type with KlDeclareType
```

2.4 Selectors declarations

New selectors MUST be declared before any application-added types are defined at runtime.

KlDeclareSelector (— ***declare a new selector***

```
int        arity,
char *     name
);
```

- `arity` — *number of arguments of associated methods*
- `name` — *external name of the selector*

Declare a new selector and returns it, i.e. the numeric index of the method.

```
/* define a new selector «rotate». The rotate method will take 2 arguments, the object itself
 * and the angle
 */
int KlSelRotate;
KlSelRotate = KlDeclareSelector(2, "rotate");
```

```
int
```
index of selector

KlGetSelectorByName (— ***returns index of named selector***

```
char *     name;
);
```

- `name` — *the name of the selector*

Returns the numeric value (index) of the selector from its printable name.

```
KlGetSelectorByName("eval")          →  10
```

KlSelectorName

```
char *                                   the printable name
(                                        return the selector name
int        selector;                     the selector itself (index)
);
```

`KlSelectorName(10)` → `"eval"`

KlSelectorArity

```
int
(                 the number of arguments of the methods called by this selector
int        selector;                     the selector itself (index)
);
```

Returns the arity (number of arguments) of the methods called by this selector. Arity are at least 1.

`KlSelectorArity(10)` → `1`

This means the C «eval» method is called with only one argument, the object itself, and the call will look like:

```
KlSend(KlSelEval, object, (object))
```

2.5 Types

2.5.1 Type declarations

Types can be declared at any time. The recommended practice is to declare as primitive Klone types all the essential data types to the application.

KlDeclareType(

declare a new klone primitive type

```
KlType *   type_pointer,     a pointer to the variable that will hold the type
char *     name,             the klone external name of the type
int        size              the size of the C structure instances of this type
);
```

Declares a new Klone type object. This function creates the type and sets its value into a C variable *type_pointer* passed by reference to it. Name will be its external name in Klone, by convention it should have its first letter capitalized. Types must be defined after all application-specific selectors have been declared.

For instance the «String» type is defined in Klone by:

```
typedef struct _KlString {
        KLONE_HEADER;                   /* common header */
        char *string;                   /* C string */
        int length;                     /* length */
} *KlString;
KlType KlStringType;                    /* in the .h */
KlDeclareType(&KlStringType, "String",
        sizeof(struct _KlString));              /* in the .c */
```

Note that **KlStringType** is the Klone standalone type object, which will be found as the first field (of C name **type**) of all C object of type string, i.e. **KlString**. Instnaces of Klone strings will thus be C objects of the C type **KlString**.

The built-in methods for a type newly declared by **KlDeclareType** are undefined, except for the following methods which have default values:

eval: default method is identity, objects evaluates to themselves
print: prints an internal form of the form **{^** *<type-name> <adress-of-object>* **}**.
equal: defaults to eq (returns t if objects are exactly the same)

hash: retruns the adress of the object as hash value
free: frees the memory block occupied by the object

KlDeclareSubType — *declare a new klone primitive type*

```
(
  KlType *   type_pointer,    a pointer to the variable that will hold the type
  char *     name,            the klone external name of the type
  KlType     parent,          the type it is derived from
  int        size             the size of the C structure instances of this type
);
```

Declares a type inheriting traits and methods from another one, **parent**, to be called after defining all the public methods of main type. Copies all traits and methods of parent type into the new type, performing thus inheritage by delegation.

```
KlDeclareSubType(&KlAtomType, "Atom", KlStringType,
          sizeof(struct _KlAtom));
```

2.5.2 Functions on types

Here are listed the public functions to access types. More are listed in the **klone.h** file for more internal use.

KlTypeName — *the name of the type as a Klone atom*

```
KlAtom
(
  KlType     type
);
```

Return a the name of the type as a Klone object. By convention, type names are capitalized (Klone is case-sensitive, unlike Common Lisp)

KlTypeCName — *the name of the type as a C string*

```
char *
(
  KlType     type
);
```

Returns a pointer to the name as stored in the Klone interpreter (in the atom describing the type name), so you should **not** modify the result, and it is **not** needed to free it.

```
KlTypeCName(KlStringType)        → "String"
```

KlTypeFatherGet — *the father of the type*

```
KlType
(
  KlType     type
);
```

Returns the parent of the type **type**, as was specified by the **KlDeclareSubType** function (see page 15), or the NULL pointer for main types (declared by **KlDeclareType**).

KlTypeNext — *access the list of all defined types*

```
KlType
(
  KlType     type
);
```

All the Klone types currently defined are stored in a linked list. The head of the list is pointed to by the global variable **Kltypes** (of C type (**KlType ***))

KlIsOfType

```
int
(
KlO         object,
KlType      type
);
```

Retuns1 if **object** is of **type** (or is of a sub-type of **type**), else 0. If type defines a trait (trait declared by the **KlDeclareIsTrait** function), returns 1 if the object has the trait.

2.6 Traits

Traits is an efficient way to express inheritance (the «is-kind-of» relation of traditional Object-Oriented world). It is an explicit mecanism, more powerful than simple inheritance. Traits define a feature that you can test for in types. Traits are implementeds as a bit that you can test in a 32-bit slot in types. For instance, traits have been defined for symbols and strings, the **KlString** type having only the string trait, and the **KlAtom** type having both traits.

The built-in traits are:

```
KlTrait_string
KlTrait_symbol
KlTrait_function
KlTrait_number
KlTrait_exo
KlTrait_table
KlTrait_hasheq
KlTrait_list
KlTrait_pseudotype
```

Traits names are stored in the **KlTraitNames** array of strings, with **KlLastTrait** being the number of already declared traits. A trait of whose name is stored as the string **KlTraitNames[N]** will thus have the value **(1 << N)**. Traits names are not used in Klone, they are just useful for debugging at the C level.

KlDeclareNewTrait — ***Allocates and create a new trait***

```
int                          New trait value
(
char *      name;            new trait name
);
```

Defines a new trait. Returns its value.

```
int KlTrait_resizable = KlDeclareNewTrait("resizable");
```

KlDeclareTrait — ***Asserts that a type has a trait***

```
(
KlType      type;
int         trait;
);
```

You must sets the traits a newly declared type bears by this call. For instance, to tell Klone that it can use safely atoms everywhere strings are expected:

```
KlDeclareTrait(KlAtomType, KlTrait_string);
```

KlDeclareIsTrait — ***This type defines this trait***

```
(
KlType      type;
int         trait;
);
```

Type is set to have the trait, but also all subsequent subtypes derievd from type will automatically bear this trait, in the classic single-inheritance mecanism.

```
KlDeclareIsTrait(KlAtomType, KlTrait_symbol);
```

```
                       int                            true or false
KlHasTrait          (                   Is object of a type having the trait?
                       KlO     object;
                       int     trait
                    );
```

Tests the presence of the bit **trait** into the type of the Klone Object **object**. Returns 0 if not.

```
                          KlO                 Returns if the error is corrected
KlArgumentMustHaveTrait (    triggers an error of type «bad argument» if object does not have the trait
                          KlO     object;
                          int     position;   The position in the argument list, staring at 0
                          int     trait;
                        );
```

Useful routine to test the validity of arguments passed to functions

```
KlO DeleteFile(KlString name) {
    KlArgumentMustHaveTrait(name, 0, KlTrait_string);
... }
```

2.7 Variable declarations

2.8 Function declarations

2.9 Utility routines

CHAPTER 3

Inside Klone

This chapter details the inner working of various parts of the Klone Engine

3.1 The Garbage Collector

3.1.1 Generalities: What is a Garbage Collector?[1]

A **memory manager** is a set of routines for managing storage in a language and/or operating system. A memory manager typically handles the problem of finding and allocating memory blocks of the appropriate size, and accepting «free» blocks back from the language and/or application for later reallocation.

An automatic memory manager or **garbage collector** is a memory manager which is automatically called by the language and/or operating system to reclaim «free» blocks when the language and/or operating system can prove that they are no longer accessible to the program and/or application.

Automatic memory managers or garbage collectors fall into two major categories: r**eference counters** and **tracers**. Reference count memory managers keep a counter for each managed object which indicates the number of «live» pointers to this object, whether from other objects, or from non-object locations called «roots», which usually consist of global variables and local variables in «stack frames» or «activation records». When the reference count of an object becomes zero, the object can be reclaimed, and its storage made available for a new usage. The reference counts of objects pointed to by this newly freed object can be immediately or promptly (strictly) decremented, leading to recursive freeing. Alternatively, the reference count decrementing of the objects pointed to by this newly freed object can be deferred or done **lazily**, at the time that the object is reallocated. This lazy reference count decrementing requires a stack or a queue for a «freelist» (in Klone this is the Zero Reference Table), but spreads the work more evenly than the «strict» reference count method usually taught.

1. Text from Henry G. Baker <hbaker@netcom.com>. Interested readers should consult by anonymous ftp the site cs.utexas.edu in directory /pub/garbage for more introductory texts

A tracing memory manager does not require the continual updating of reference counts, but occasionally traces out the entire «accessibility» relation by starting from the «roots» and computing the transitive closure of the «points to» relation. After this has been done, the GC can now be sure that the «complement» of the set of accessible memory locations is thus «inaccessible» and can be reused for other purposes. There are generally two methods of tracing -- **marking** and **copying**. Marking is the most used method in traditional lisp systems and simply paints each object as it sees it, and recursively paints the objects it points to, and when everything that is accessible has been painted, it knows that unpainted objects are inaccessible and therefore garbage. Copying is even simpler; as each accessible object is found, it is moved to a new location, leaving a forwarding address. After all accessible objects have been moved, the old neighborhood is simply condemned en masse.

Thus, all automatic memory managers utilize the concept of a «root», which is a non-object pointer to an object. Traditionally, languages which utilized automatic memory managers had their compilers provide detailed information about the «roots» that the memory manager needed. A reference counting manager needs to know when a root is created, and when one goes away. A tracing memory manager needs to be able to «enumerate» (iterate through) the roots.

3.1.2 Klone GC Principles

reference counting

The Klone Garbage Collector is implemented upon the old technique rarely seen nowadays in lisp-like systems, «reference counting». Although the main drawback of this method is being unable to automatically free circular lists, it provides a very simple and portable way to perform incremental Garbage Collection occuring at a steady and regular pace during computaion as opposed to the prevalent modern mecanisms of «mark and sweep» or «stop and copy», which allows interactive systems not to be stopped while a GC is performed, as in emacs. Moreover it was felt that as used as a customisation language, the programs that were to be written in Klone were simpler than say Common Lisp ones, and thus the burden to avoid using circular structures was a reasonable one, considering a steady operation was provided. And to be seamlessly integrated with C, it should preserve the notion of pointers, and thus not move objects around like the «stop and copy» method.

explicit roots tracking

For portability, Klones relies on the C programmer to take care of «roots tracking», that is explicitely incrementing and decrementing the reference count of objets as they are pointed to and de-referenced by «roots», i.e. global variables or fields of global structures, or structures belonging to the non-Klone part of the application.

For instance, if you have a «widget» strcuture and in it a field «klone_callback» used to hold a pointer to a Klone expression that you want to evualte in response to some event, storing a klone expression `expr` in this field should be done by the code:

```
KlDecRef(widget.klone_callback); /* decrement count of previous value */
KlIncRef(widget.klone_callback = expr); /* set and increment new value */
```

Note that `widget.klone_callback` can be the NULL pointer 0, as `WlDecRef` does nothing on Null pointers. `WlDecRefNonNull` can be called when you are sure the pointer is always valid as it is a bit faster than `WlDecRef`.

Of course, this is to be done only on pointer affectations done by the programmer himself. Klones manages automatically all its internal «roots».

delayed scope garbage

The other traditional drawback of Reference counting GC methods was its alleged slownessas objets reference count were constantly incremented or decremented. This was avoided in Klone by realizing that interactive systems were built around a «main loop» calling functions to be executed and the result of this functions being always discarded. This allowed me to build a innovative «delayed scope garbage» scheme saving a lot of increment/decrement operations and providing a predictible operation from the point of view of the C programmer.

On creation, the reference count of an object is set to 0, and when an object reference count drops to 0 via the `KlDecRef` macro, or at creation, the objet is stacked onto the `KlZrt` (Klone Zero Reference Table) if it is not already there. Then, on explicit calling of the GC (via `KlGC()`), all the objects in the `KlZrt` which are still at refcount 0 are freed by calling their internal «free» method (associated to the Klone selector `KlSelFree`).

But this cleanup phase is «scoped», i.e. it only examine objets that were stacked after the current Marker has been set for the current C scope by `KlGCMark()` (in the variable declaration section of C blocks). This technique ensures that objets belonging to an outer scope will never be freed by any action of an inner scope and avoid one traditional drawback of mixed lisp/C programming of having to always «grab» temporary objects to prevent them from being freed under your feet by the GC that can happen at any time.

A typical Klone C code would thus look like:

```
{
    KlGCMark();           /* protects outer scope from GC */
    do_some_things();
    KlGC();               /* free garbageable objets from this scope */
}
```

NOTE: This means that the GC can only be called on «closed» scopes, that is functions with no Klone value returned, as returning a value moves it in fact in the outer scope, in «unprotected» space where it could be destroyed before being returned. So KlGC() should NOT be called in a functional scope returning a value.

This code is illegal:

```
KlO foo() {
    KlO obj;
    KlGCMark();
    obj = do_some_things();/* creation of obj */
    do_some_other_things();
    KlGC();
    return obj;           /* is ILLEGAL ! */
}
```

But this one is valid, since the object `obj` began its existence outside of teh GC scope:

```
KlO foo() {
    KlO obj = do_some_things();
    KlGCMark();
    do_some_other_things();
    KlGC();
    return obj;           /* Ok, as obj was created in an outer scopeLLEGAL ! */
}
```

3.1.3 Implementation

Each Klone object has two mandatory fields at its beginning, a pointer to its type (field `type`) and reference count info (field `reference_count`), a 32-bit integer actually consisting of a 31-bit number, the reference count itself, and a bit (lowest one) used for internal purposes (keeping track whether the object is already on the list of to-de-deleted objets, the «Zero Reference Table»). So to get the actual reference count of an object, you must divide the contents of the C field `reference_count` by 2. The remaining of the division indicates (if 0) whether this object is already in the KlZrt.

K
KlArgumentMustHaveTrait 16
KlDeclareIsTrait 15
KlDeclareNewTrait 15
KlDeclareSelector 12
KlDeclareSubType 14, 14
KlDeclareTrait 15
KlDeclareType 13, 16
KlGetSelectorByName 12
KlHasTrait 16
KlLastTrait 15
KlSelectorArity 13
KlSelectorName 13
KlTraitNames 15
KlTypeCName 14
KlTypeFatherGet 14
KlTypeName 14
KlTypeNext 14
Kltypes 14
R
reference_count 11
T
type 11