Object Management Conventions and Coding Support for `mylib`

Gene Myers

July 28, 2009

The `mylib` library supports fairly complex object classes such as multi-dimensional arrays, level-set trees, and water-shed decompositions of an image, that we think necessitate an object-oriented style despite the fact that C is the implementation language. Therefore we have developed and consistently follow a set of object management and memory usage conventions that are adhered to throughout the library.

The conventions were chosen with an eye towards optimizing performance (we presume you would be writing in something like Perl or Matlab if you didn't need the outmost in speed and memory utilization). The dynamic memory allocation required for object creation and destruction is provided by the `malloc` suite and occurs in the system heap. Anybody that has ever implemented a heap knows there can be substantial time and memory overhead whenever excessive fragmentation occurs and such fragmentation is inevitable in any long running or complex application. Therefore we always try to minimize the use of the heap. If you don't believe us, try doing a million `malloc`'s followed by a million `free`'s of those `malloc`'d blocks in some random order. The `malloc`'s are fast but the ensuing `free`'s will cost you seconds of CPU time. Therefore, a particular emphasis of our object management scheme is enabling a user to have fine-grained control over object memory usage and to adopt conventions that allow for highly efficient memory management.

The first section is required reading for all users of `mylib` as it describes the essential conventions and generic routines for every class. The second section should be read by those wishing to contribute code to `mylib` as it describes the macro support that makes it easy to realize an implementation of the conventions for any class you might develop. The third section carefully documents the code generated in response to the macro support described in the second section. It serves as the documentation for the custom preprocessor that realizes the macro transformations.

1 Object Management Conventions

1.1 Convention 1: Object Creation and Destruction

Every data abstraction or class has routines to create, copy, and destroy objects of the class. For example an `Array` object is modelled by the following record and, among (many) others, has the four routines given below:

```
typedef struct
  { Array_Kind  kind;    //  Interpreation of the array: one of the four enum constants above
    Array_Type  type;    //  Type of values, one of the eight enum constants above
    int         scale;   //  # of bits in integer values

    int         ndims;   //  Number of dimensions of the array
    uint64      size;    //  Total number of elements in the array (= PROD_i dims[i])
    int        *dims;    //  dims[i] = length of dimension i

    char       *text;    //  An arbitrary string label, '\0'-terminated as per C convention

    void       *data;    //  A block of size sizeof(type) * size bytes holding the array.
  } Array;


Array *Make_Array(Array_Kind kind, Array_Type type, int ndims, int *dims);
Array *Read_Image(char *file_name, int layer);
Array *Copy_Array(Array *array);
void   Kill_Array(Array *array);
```

The routine `Make_Array` creates a new array object and returns a pointer to it. So does the routine `Read_Image`. Such routines we term *generators* and often a class will have more than one. The user is responsible for managing the newly created object including the memory it occupies. When the object is no longer needed its storage should be returned to the system heap with a call to `Kill_Array`. If another copy of an object is needed, it can be created with `Copy_Array` which is also considered to be a generator for the class.

Note that for an `Array` object the user is shown the internal fields of the top-level record of the object. The convention is that a user is free to examine these fields but should shy away from ever setting them unless they really understand the impact thereof. We don't over-protect the user and often, when we think it useful, reveal the fields of an object. However, when an object's encoding is so complex that it is unlikely to be of any direct value to the user, we hide the type of object by declaring it to be of type `void`. For example, a level set tree object (that models the tree of level sets of an image) has, among others, the following declarations:

```
typedef void Level_Tree;

Level_tree *Build_Level_Tree(Pixel_APart *image, int iscon2n);
Level_tree *Copy_Level_Tree(Level_Tree *tree);
void        Kill_Level_Tree(Level_Tree *tree);
```

In summary, every class has a least one generator besides its Copy-routine, and the Copy and Kill routines always have the form `<X> *Copy_<X>(<X> *)` and `void Kill_<X>(<X> *)` where `<X>` is the class name.

1.2 Convention 2: Recycling and Buffering Objects

Thus far our management of objects is quite conventional: a generator acquires memory for an object it is creating from the system heap, and the corresponding Kill routine returns the memory to the heap. But often applications use only one or a small handfull of objects of a given type at any one moment. For example, as shown below, one might repeat a for-loop a thousand times and in each iteration create an Array by reading an image file, analyze the image in some way, and then kill it. Only one Array object is in existance at any given moment, but a thousand system allocations and frees are performed.

```
for (i = 1; i <= 1000; i++)
  { Array *image = Read_Image(...);
    // analyze the image for something cool
    Kill_Array(image);
  }
```

To alleviate this wasted effort, especially in the common case where every image is of the same dimensions, every class has a routine `void Free_<X>(<X> *)` that effectively recycles the object by placing it on an internal recycle list from which a generator will grab a previously used object if one is available, rather than malloc a new one. The recycle list is always realized as a last-in, first-out stack. Returning to the example above, if one frees the Array created in each loop, as opposed to killing it, i.e. replaces Kill_Array with Free_Array, then only one Array will ever be allocated from the system heap, and it will be reused in each iteration.

Many objects, such as an Array, have one or more fields that point to blocks of memory whose size can vary depending on the value of the object, e.g. dims whose size is proportional to the dimensionality of the array, text whose size is that of its \0-terminated label string, and data which depends on the kind, type, and dimensions of the array. Obviously when a generator picks up such an object from the recycle list, one or more of the blocks may not be large enough to accomodate the value(s) of the object the generator is being asked to make. In this case, those blocks of the object that are not large enough are expanded with calls to realloc. Obviously the class must know the size of these memory blocks. These are contained in a *wrapper* that surrounds the object record but is always hidden from the user. If you are interested see Section 3 for details of how this is implemented. In the event a block is bigger then necessary, then it is not reduced in size, but left as is with the excess part of the block unused. So in the case of our running example, the data-block of the Array that is being recycled in each iteration, will grow to the size of the largest Array required in any iteration. Probabilistically, a reallocation will not be necessary very often if the block sizes follow a distribution with exponentially vanishing tails.

Indeed by theory, the number is expected to be on the order of the logarithm of the number of iterations. To further reduce reallocations a generator may anticipate future growth in object sizes by, for example, reallocating blocks by 10-20% more than is needed, or keeping track of the largest object they have been asked to produce and and always asking for a block of this largest size, rather than of the needed size.

This *buffering* reduces the number of reallocations that are required at the expense of using more space than is absolutely necessary for any given object. However, in the example, where only one object is ever extent, this overhead is more than compensated for by the reduction in usage of the system heap. However, in cases where many instances of an object type will *persist*, or the objects are so large that any overhead in their memory blocks is problematic (e.g. a 400 mega-pixel image stack), one can *pack* the object by calling the routine `<X> *Pack_<X>(<X> *)` on each object. The effect is that the space for the memory blocks of an object are reduced to exactly the sizes needed for the object's value by calling `realloc` as necessary, and as a convenience a pointer to the object is returned.

Over time the memory blocks for such objects, if recycled through the use of the `Free`-routine, become larger than needed for the value of the object currently occupying them. While, as suggested above, some generators further pad the size of the blocks by keeping and requesting a highwater-mark size and further geometrically expanding it, this is a design choice that can vary from class to class. Indeed, for very large objects, such as 3D image stacks, padding is *not* a good idea. For example, the generators for arrays pad the size request for small arrays, but if the needed size is very large then no padding is requested. Moreover, if a memory block of an object shell taken from the recycle list is very big and much larger then what is needed by the object being generatored, then in fact the generator packs it automatically. While the specific buffering policy of a class is hidden from the user and is at the discretion of the implementor, it is always the case that the user can pack objects when the effect of any such buffering policy is deemed detrimental to the application.

As a final comment, notice again that for our running example, often all the images are of exactly the same size, in which case no reallocation takes place at all. A single array is allocated in the first iteration and then efficiently reused thereafter! We conclude with a revised outline of our running example, in which the single array holding the image is reused and then returned to the system heap after the last iteration.

```
for (i = 1; i <= 1000; i++)
  { Array *image = Read_Image(...);
    // analyze the image for something cool
    if (i < 1000)
      Free_Array(image);
    else
      Kill_Array(image);
  }
```

1.3 Convention 3: Resetting and Monitoring Classes

The recycling and buffering scheme of Convention 2 is designed to permit the user, who generally has some *a priori* knowledge of the usage pattern of his application, to avoid unecessary allocations and re-allocations of memory and objects. But we foresee that in large or long-runnning applications, different object regimes may be present during different phases of a computation or an end-user's work flow. For example, an application may first do some work on some 1024×1024×256 3D image stacks, and then later do some work on a large number of 512×512 2D images. In such a case, one would ideally like to restart the buffering process and not use containers that are 1024 times bigger than necessary! Therefore, for every class `<X>` we provide a routine of the form `void Reset_<X>()` that has the effect of (1) emptying the recycle list and killing every object on it, (2) returning any working storage to the system heap, and (3) reseting every buffer size to 0. That is, the state of the class is restored to its original condition at the start of execution, save for any objects that the user may still be holding.

Any well-developed code should not contain any memory leaks, i.e., it should never lose track of an object and so fail to kill or free it when it is no longer needed. To assist users in testing this, for each class we provide a routine `int <X>_Usage()` that returns the number of objects of type `<X>` that the class has given to the user but that have not as yet been freed or killed. A typical usage would be to check that the value returned is 0 at points in your computation where you believe that all objects of type `<X>` should have been killed or are sitting on the recycle list (which can always be emptied with a call to `Reset_<X>`).

In addition, we provide a routine `void <X>_List(void (*han)(<X> *))` that calls a user-supplied handler routine `han` with a pointer to each object of type `<X>` that is currently in use by the user. One might, for example call this routine with a handler that prints out information about each object in use, including its reference count (see the next section), as a debugging aid. Another possibility would be to use a handler that frees or kills every object in use, thus guaranteeing that all memory is freed.

1.4 Convention 4: References Versus Sub-Objects

A class object consists of a record that can have pointers to several blocks of memory (that may in turn refer to other blocks of memory albeit rarely). In addition an object record can have pointers to objects of other classes that are considered part of the given object. For example, a `Tiff` object has its type hidden from the user but within its defining code module it is realized as a `Tio` record which has pointers to a `Tiff_Reader` object and a `Tiff_Writer` object. These objects are considered *sub-objects* of the `Tiff` object in that they are freed, killed, packed, or copied whenever the `Tiff` object is freed, killed, packed, or copied.

```
typedef void Tiff;             // Realized internally as a Tio

typedef struct
  { Tiff_Reader *reader;
    Tiff_Writer *writer;
    int          eof;
  } Tio;
```

On the otherhand a `Level_Tree` object's hidden `Comtree` record has a pointer `apart_ref` to an `APart` object which is the array or slice from which its level sets were determined. In this case one clearly doesn't want the `APart` treated as a sub-object of the `Level_Tree` object, but rather we simply want a *reference* to the image. That is, other derived objects such as a watershed partitioning, or a collection of regions may all want to refer to the, possibly large, source image. It would be an implementation disaster to make numerous copies of the image!

```
typedef void Level_Tree;   // Realized internally as a Comtree

typedef struct
  { APart  *apart_ref;   //  Reference to the image or slice level tree was made from
    vertex *level_tree;  //  Memory block containing the vertices of the tree
    int     iscon2n;     //  Connectivity used in building the level tree
    vertex *regtrees;    //  Binary tree of islands = level_tree-1;
    int     csize;       //  Size of tree (also of underlying APart)
    bool    is8;         //  Array is 8-bit (versus 16-bit)
    uint8  *value8;      //  Pixel values of apart (undefined if is8 == 0)
    uint16 *value16;     //  Pixel values of apart (undefined if is8 != 0)
  } Comtree;
```

If an object is to have more than one reference from several different objects, then the object must keep track of how many there are in a *reference count*. This reference count is hidden in the wrapper that surrounds each object. It is initially set to 1 by the generator that creates the object and this reference can be thought of as being given to the user. From there routines can create additional references by calling the routine `<X> *Inc_<X>(<X> *)` that increments the count by 1 and returns a pointer to the object in question as a convenience. This reference count is decremented by 1 whenever `Free` or `Kill` is called upon its object, and when, and only when the reference count reaches 0 does the object in question truly get freed, i.e. returned to the recycle list, or killed, i.e. returned to the system heap, depending on whether `Free` or `Kill` was called.

We adopt the convention of adding the suffix `_ref` to any field that is a reference to another object, as opposed to a pointer to a sub-object of the current object. An object R referred to by such a reference field is not copied or packed when the refering object O is copied or packed, but when O is freed or killed then R does have a free or a kill called upon it. The trick is that when the reference is first assigned to the field of O during its generation, the `Inc`-routine is called on R so that R gets an extra reference count. So later when O is freed or killed, the free or kill on R decrements that extra count and so R stays in existence, unless of course, the user has freed or killed their reference to R in the interim since generating O. Notice that the nice thing about this scheme is that the user doesn't need to worry about whether the referred object is still extant, it is guaranteed to be until all references have disappeared.

The `Comtree` record is hidden from the user, so how does the *user* know that when a `Level_Tree` is built an extra reference to its source image is created? First, this is always documented in the comments in the include file containing the declaration for the generator `Build_Level_Tree` that builds the tree. Secondly, wherever a declaration for `Build_Level_Tree` occurs, the `image` parameter will be annotated (see Convention 6) as receiving an additional reference, specifically:

```
Level_tree *G(Build_Level_Tree)(Pixel_APart *I(image), int iscon2n);
```

The identity `I()` macro wrapped around `image` is our convention for declaring that `image` will have `Inc_Array` called on it, and the `G()` macro indicates that `Build_Level_Tree` is a generator of `Level_Tree` objects. More on this in Convention 6. Finally, one can always get the current reference count for an object by calling the routine `<X> *<X>_Refcount(<X> *)`.

1.5 Convention 5: Bundles For Multi-Valued Parameters and Function Results

It is often the case that the computation of a given function creates multiple outputs. For example, we want a routine `Array_Range` to take an array as input and return the minimum and maximum values in the array as double precision numbers. Conceptually this pair of numbers consitute a range that conceptually we would like think of as *the* result. But a function in C can only return a scalar value so the only way we can accomplish this is to return a pointer to a structure `Range_B` that has a field for the minimum and maximum:

```
typedef struct
  { double maxval;
    double minval;
  } Range_Bundle;

Range_Bundle *Array_Range(..., Array *array);
```

One is immediately faced with the issue of where the `Range_B` record resides. If the routine creates it on the system heap then at some later point the space will need to be freed. That immediately suggests that one make it a proper object so that its storage management conventions are understood and uniform across the system. But making an object class just to return a compound result seemed excessive to us, especially since in cases such as this one the result will likely be short-lived. So instead, we ask that the user supply us a pointer to a record of type `Range_B` and the routine fills it out and returns a pointer to it for convenience. (In an earlier incarnation of the code the object was static to the routine, but we moved away from this as it does not permit re-entrant code, more on this in Section ..). We call this a *bundle* and we distinguish such records by always ending their name with the suffix `_Bundle`. We sketch the template just described:

```
Range_Bundle *Array_Range(Range_Bundle *range, Array *a)
{ ...
  range->minval = ...;
  range->maxval = ...;
  return (range);
}
```

The limitation is that the values of the bundle will be reset the next time the routine is called, but this is fine with us as the idea is that a bundle is a short-lived object. Moreover, if you need a bundle to persist a while, then you can assign it to a variable of that type that is itself either static or on the system stack. To illustrate we give an artifical example of a routine that returns the intersection of the ranges of two arrays:

```
Range_Bundle *Common_Range(Array *a, Array *b, Range_Bundle *range)
{ Range_Bundle temp;

  Array_Range(range,a);
  Array_Range(&temp,b);
  if (temp.minval > range->minval)
    range->minval = temp.minval;
  if (temp.maxval < range->maxval)
    range->maxval = temp.maxval;
  return (range);
}
```

Another use for a bundle record is to pass in as an argument a short-term collection of related values. For example, the drawing routines need to be given a paint brush which specifies a drawing operator and a “color” that matches the kind of the array. We give the declarations of the three types of color bundles below, along with that of a `void Brush_Bundle` which can be any one of the three types, and a typical drawing routine all of which take a “canvas” `Array` and `Brush_Bundle` as input, among other parameters.

```
typedef struct
  { Drawer     op;
    double     val;
  } Plain_Bundle;

typedef struct
  { Drawer     op;
    double     real;
    double     imag;
  } Complex_Bundle;

typedef struct
  { Drawer     op;
    double     red;
    double     green;
    double     blue;
    double     alpha;
  } Color_Bundle;

typedef void Brush_Bundle;
```

```
void Draw_Level_Set(Array *canvas, Brush_Bundle *brush, Level_Set *r);
```

The drawing routines know the kind of the canvas `Array` and so always know what kind of specific paint brush to coerce the `Brush_Bundle` parameter to. For example:

```
static Color_Bundle yellow_overlay = { MIN_PIX, 1., 1., 0. };

Draw_Level_Set(canvas,&yellow_overlay,r);
```

In summary, a bundle is a record, or more consisely a `struct`, that is used as a means of communicating multiple values to and from a routine. The storage for a bundle is always managed by the user and is most typically on the system stack, especially for applications where code needs to be re-entrant. It is not an object, its purpose is to pass in conceptual related values as an argument, or to receive conceptually related values as a function output.

1.6 Convention 6: Parameter and Return Value Annotations

The library has so many routines that its pretty much impossible to remember them all yet alone remember the parameter usage conventions for each. So a handy index is available that lists by module every routine, type, and constant, and in addition uses a simple annotation scheme to indicate the input/output usage of every parameter and more. The scheme consists of a set of one capital letter identity macros that are placed around a parameter or function name, where the letter indicates the role(s) of the parameter or function name. As an example, we list a couple of macros and give a fictitious function declaration:

```
#define M(x) x  //  parameter x is (M)odified by the routine, i.e. input & output
#define O(x) x  //  parameter x is set by the routine, i.e. (O)utput only
#define G(x) x  //  function is a (G)enerator

Array *G(Example)(Array *M(work), int *O(status));
```

First note that after macro expansion the declaration is just

```
Array *Example(Array *work, int *status);
```

so the annotations basically disappear before compilation. Indeed we use them in the code and associated include files to enhance the documentation of interfaces. In the example above the annotations indicate that the function `Example` is a generator, that the argument `work` is modified by the routine, and that the integer pointed at by `status` is set by the routine.

A parameter to a routine can provide a value to the routine as input, it can return a value from the routine as output, or it can serve as both. We use the letter `O` to denote strict output, and the letter `M` to denote modification, i.e. both input and output. By far most parameters are used as input and not modified, so our convention is to not mark such parameters, i.e. no `M`- or `O`-annotation implies the parameter is strictly an input.

We also annotate return values of functions. A function that is a generator has its function name annotated with the letter `G` (for *g*enerator). It is also very common that a routine will return a pointer to one of its arguments, which we indicate by annotating the parameter with the letter `R` (for *r*eturn value). The only other possibilities are that a function returns (a) a scalar value (e.g. `int` or `double`), or (b) a pointer to a component of an object, or (c) a pointer to a vector of scalars (e.g. `int *` or `double *`). Case (a) clearly requires no annotation. In case (b) we effectively have a return result that (1) is short-term in the sense that another call to the function or an object management call on a relevant object can void the value, and (2) has its memory management being taken care of by someone other than you. Because case (c) would not be re-entrant, we avoid this case, it does not occur in our library. We give the following real examples:

```
Array *G(Make_Array)(Array_Kind kind, Array_Type, int ndims, int *dims);

Array *Threshold_Array(Array *R(M(a)), double cutoff);

Range_Bundle *Array_Range(Range_Bundle *R(O(range)), Array *array);

int *Collapse_Watershed(Watershed *shed, void *base, int size,
                        int (*handler)(void *,int,void *), int *O(newcbs));
```

The function `Make_Array` is a generator all of whose arguments are strictly input. `Threshold_Array` takes an array `a` as input, modifies it, and returns a pointer to it as its result. Note that `a` is annotated with both `R` and `M`, a frequent case, by simply nesting the annotation macros. `Array_Range` clearly returns a pointer to a bundle that is passed to it and whose values are set by the routine. By deduction `Collapse_Watershed` returns a vector of integers and sets the integer pointed at by `newcbs`. The integer vector is guaranteed to persist until the next call to the routine or until `Reset_Watershed` is called. Note as a general principle that only one `R` can occur, and if it is present, then `G` does not occur.

Our object management conventions involve sub-objects versus references, and frees versus kills. So we further carefully annotate events involving these phenomenon. In particular if a parameter is freed by a routine then it is annotated with an `F`, if it is killed it is annoted with a `K`, and if a new reference to it is created then it is annotated with an `I` (for *i*ncremented). A more subtle phenomenon is that some generator routines take an argument object and reference it within the object they are making, or even more strongly, take the argument object and make it a sub-object of the object they are making. In the later case we annotate the argument with the letter `S` (for either *s*ub-object or *s*ubsumed) and the former case with the letter `C` (for *c*onsumed). In the case of being consumed one can think of it as an increment and free combined; other parties can reference the object but the reference you passed in has been taken over by the generated object. In the case of being subsumed, the generator routine more strongly checks that this is the *only* reference to the object as the generated object is going to completely take it over, and may even modify the argument object as it pleases. We give some real examples:

```
Coordinate *G(Coord)(char *list);
Coordinate *AddCoord(int d, Coordinate *R(M(coord)));
int         Coord2IdxA(Array *array, Coordinate *F(coord));

XCanvas    *G(Begin_Xfig_Drawing)(char *name, XTransform *F(xform));
void        Finish_Xfig_Drawing(XCanvas *K(canvas));

Level_Tree *G(Build_Level_Tree)(Pixel_APart *I(image), int iscon2n);

LU_Factor *G(LU_Decompose)(Double_Matrix *S(m), int *O(stable));
```

An integer coordinate vector is typically a short-lived small object that allows one to conveniently produce indices into arrays. The generator `Coord` creates one, the routine `AddCoord` modifies one by adding an extra index, and the converter routine `Coord2IdxA` converts `coord` into an index into the flat data space of `Array array`, freeing the input coordinate when it is done. The xfig drawing module allows one to start a canvas by creating an `XCanvas` object with `Begin_Xfig_Drawing` and then later when you finish the drawing with `Finish_Xfig_Drawing` the `XCanvas` is killed. When one generates a water shed object with `Build_Level_Tree` the image for which it is built gets referenced by the new object. Finally, when one produces an LU-decomposition of a square matrix `m`, the matrix is modified and becomse a sub-object of the generated `LU_Factor` object. Note carefully the difference in the fate of the two arguments in these last two examples. The image is still yours after building the watershed, a new reference is created and given to the watershed object. But the matrix becomes part of the `LU_Factor` object, and if you had, more than one reference to the matrix an error would have been flagged.

1.7 Convention 7: Read and Write routines

Most object classes have routines that read and write an instance of an object of the class from or to a file. The write routine has the specification `void Write_<X>(<X> *x, FILE *output)` where `<X>` is the class name, and the read routine is a generator of the form `<X> *Read_<X>(FILE *input)`. On output, sub-objects of a class are automatically written as part of the write of the object, where as references to another object are *not*. Thus on input, the read routine sets an object's references to `NULL`. The user must subsequently re-establish any such references using routines that should be provided by the object class for this (and other) purposes.

In some exceptional cases it doesn't make sense for a class to have a read and write routine, in which case they don't. For example, the object classes `Tiff` and `XCanvas` don't have such routines as they are each controlling an input/output process. The value of these objects is a function of the file they are reading or writing and as such one can't really record or recreate the object's value unless one went to the ridiculous length of re-establishing the state of the I/O process.

Objects may be read and written sequentially from a file. Each reader assumes that the input cursor of the file is at the same position the write of an object of its type began. The reader is guaranteed to move the cursor to the beginning of the next object on the input stream. Moveover we take the extra precaution of beginning the write of an object by outputting the ASCII name of the object class, e.g. `"Array"`, at the start so that (a) a user has the ability to see what objects are in a file (even though the rest of the file is coded in binary), and (b) so that the read routine can verify that it is trying to read an object of the class.

2 Memory Management Support

The copy, pack, inc, free, kill, reset, read, and write routines for each data abstraction or class are stylistically so similar that I decided to build a preprocessor that given a few parameters about the class, automatically generates most of the C code realizing the required operations. The preprocessor is of a complexity beyond the capabilities of the standard C preprocessor, so I wrote a custom preprocessor in C to implement the desired transformations including the initial allocation of an object, the sizing of any memory blocks, and maintenance of the reference count if present. This complicates compilation of a source file in that first one must run the memory management preprocessor on the file and then compile the resulting C code. The simple `makefile` that compiles the library of all modules for the `mylib` library clearly illustrates how this is done should you want to write your own module and add it to the build.

A single line in a file of the form that begins with `MANAGER` followed by a number of parameters will be expanded into code that when coupled with a little additional code tailored for the class will realize the required memory management routines for the class. The specific syntax is as follows:

‘`MANAGER`’ `[-cpkfr[iI][oO]]` <class names> <field descriptor> $*$

<class names>	→	<visible class name>(‘(’<hidden class name>‘)’)?
<field descriptor>	→	<block field>‘:’<wrapper size name>
	|	<block field>‘!’<64-bit wrapper name>
	|	<subobject field>‘*’<subobject class>
	|	<reference field>‘@’<reference class>

The keyword `MANAGER` must begin at the far left of its line, and is followed first by some optional flags, then a class name parameter, and then zero or more field descriptors, all of which must be separated by white space. The class name parameter gives the name of the class visible to the user, and if this hidden, i.e. declare to be of type `void`, then one further gives within parenthesis the type name used within the class module to define the top-level structure of the object. In the case that the object has pointers to sub-objects or to variable-sized component arrays, then a field descriptor for each needs to be given to the preprocessor. Each field descriptor consists of the relevant field name in the top-level structure followed by a punctuation mark — either : , !, *, or @ — followed by a unique name to be used in the object wrapper in the case of a 32- or 64-bit memory block field, the type name of a sub-object, or the type name of a reference, respectively (See Convention 4).

In response to a `MANAGER` specification line, the object management preprocessor replaces it with

1. a type declaration for the object wrapper and global variables for the class
2. code for `Copy_<X>`, or `copy_<i>` if the `-c` flag is set
3. code for `Pack_<X>`, or `pack_<i>` if `-p` flag is set

4. code for Free_<X>, or free_<i> if the -f flag is set
5. code for Kill_<X>, or kill_<i> if the -k flag is set
6. code for Reset_<X>, or reset_<i> if the -r flag is set
7. code for Write_<X> if the -O flag is set, or write_<i> if the -o flag is set
8. code for Read_<X> if the -I flag is set, or read_<i> if the -i flag is set
9. code for Inc_<X>, <X>_Refcount, <X>_Usage, and <X>_List
10. for each memory block, the following two routines are generated:

    ```
    static inline int allocate_<i>_<block_field>(<I> *<i>, (uint32|uint64) <size_name>,
                                                      char *routine);
    static inline (uint32|uint64)  sizeof_<i>_<block_field>(<I> *<i>);
    ```

11. and a routine allocates a new object:

    ```
    static inline <I> *new_<i>((uint32|uint64) <size_name>, ..., char *routine);
    ```

where <X> is the visible class name and <I> is the hidden class name or the visible class name if a hidden name is not given. Moreover, <x> and <i> are all lower-case versions of <X> and <I>, respectively. For example, if <X> is Array, then <x> is array. Note that we are assuming by our coding conventions that <X> and <I> are capitalized.

Note in items 10 and 11 that the type of some parameters or return values can be either uint32 or uint64. If you declare a memory block with a : as the separator then the memory block is assumed to by less than 4Gb and uint32 is used as the type of its block field. But if ! is used as the separator, then one is designating that a memory block can be very large and a uint64 is used as the type where appropriate.

Every routine produced by the object manager calls the simple routine Guarded_Realloc in utilities.h whenever it needs a new memory block or whenever it needs to resize one. Guarded_Realloc as written, either does the reallocation (it calls realloc) or it prints an error message and stops the program. This behavior is fine for batch-oriented command-line programs (the main intended use for mylib), but it is not so in other possibly desired scenarios, for example, as subroutines in an interactive user-interface. To change the behavior, simply re-implement a static version of Guarded_Realloc in the relevant module(s), so that it returns NULL if it can't perform the reallocation. All the routines produced by the object manager are designed to work in this scenario and will either return NULL to signal they failed (new_<i>, Pack_<X>, read_<i>, and Read_<X>), or a non-zero value if they failed (allocate_<i>_<block_field> and pack_<i>).

The creation routine new_<i> is used by every generator (including the copy routine the manager creates), to get a new object, either from the recycle list, or from the system heap if necessary. The parameters <size_name>, ... are the requested sizes for the memory blocks in the field descriptor list, in the order given. The parameter routine is the routine name to report should an out-of-memory error occur when new_<i> requests memory from the system heap, assuming the typical behavior of Guarded_Realloc (see paragraph above). If Guarded_Realloc is re-programmed to not stop the program, then new_<i> will return NULL if it can't allocate the object as requested.

The routine allocate_<i>_<block_field> has the effect of making sure that <block_field> points to a block of memory equal to or larger than the requested size. If the current block isn't large enough then it increases its size, or returns a non-zero value if it can't and Guarded_Realloc hasn't stoped the program. Incidentally, new_<i> calls this routine for each memory block in the object. The routine sizeof_<i>_<block_field> returns the padded size of the memory block pointed at by <block_field>. This allows a generator to realize a buffering scheme that knows when the padding for a block has become excessive.

All the fields that point to sub-objects or references are set to NULL by new_<i>. The generator that calls this routine will presumably set the values of these fields along with all the others. Moreover, you can pass 0 as a size to new_<i>, in which the relevant memory block does not get allocated and the field pointer is to NULL. This allows you to do the allocation/resizing of a memory block with its appropriate allocate_<i> routine later. This is especially useful when the size of a block isn't known until you are further into the generation of the object. Lastly, care has been taken that the codes for all the operations generated by the object manager are careful to not manipulate a pointer field whose value is NULL. Moreover, a size of 0 can be assigned to a memory block by calling its allocate_<i> routine with 0 as the size. If the given object is subsequently packed, then that memory block is returned to the system heap and its field is set to NULL. In summary, if you are careful, you can realize objects in which fields can be undefined if desired.

While the MANAGER macro generates a great deal of the required code, it may still need a little extra code from you. First, if the object has memory blocks then for each one you need to supply a routine of the form:

```
static inline (uint32|uint64) <i>_<size_name>(<I> *<i>)
```

that returns the size in bytes of the value sitting in the memory block. Note carefully that this is the actual size of the used portion of the block, the manager already knows the padded size, and indeed will give it to you with a call to the appropriate sizeof_<i> routine described earlier. The return type depends on whether you have declared this to be a regular block or a 64-bit block.

Based on the information you've given it, the manager may not be able to produce code that does everything that you might like your copy, pack, free, kill, reset, read and write routines to do. For example, the manager can only produce code for a reset that kills and removes all objects on the recycle list. But you may also wish to reset any buffering size limits and free any working storage used by the class. For each of these seven routines setting the appropriate flag produces a *core* routine (e.g. copy_<i> if the -c option is set) as opposed to an external routine (e.g. `Copy_<X>` as described in items 2 through 8 above. If such a core routine is requested then the implementor must write the external routine. This external routine must call the core routine as well as perform the additional functions the implementor desires. Note that in the case of read, an external routine is produced only if the `-I` is set, and if neither the `-i` or `-I` flag is set then *no* read code is produced. The same is true of write as from Convention 7 a class need not always have read and write routines. We list each core routine declaration followed by a description of what precisely the routine does and suggest when you might need or want to supply the external routine.

`static inline <I> *copy_<i>(<I> *<i>);`

> This routine makes a copy of the object record, assigns each block field of the copied record to a copy of each memory block, assign each sub-object field of the copied record to the result of calling each sub-object's copy routine, and increments each reference. If a memory block contains information, such as pointers, that are relative as opposed to absolute, then you need to write the external `Copy_<X>` routine and after calling the core routine adjust the relative information.

`static inline int pack_<i>(<I> *<i>);`

> This routine packs each of the object's memory blocks and calls each sub-object's pack routine on the sub-object. The circumstances requiring the writing of the external routine are the same as for the copy routine because when a *realloc* is called on a memory block, it might be moved to a new location by the system heap.

`static inline void free_<i>(<I> *<i>);`

> This routine decrements the object's reference count and returns if it is not zero. Otherwise the routine puts the object on the recycle list, calls each sub-object's and referenced object's free routine, and adjusts a global counter of how many objects are currently in use. If a memory block contains pointers to other objects or memory blocks, then you need to write the external free routine in order to free or kill those objects and blocks.

`static inline void kill_<i>(<I> *<i>);`

> This routine first decrements and checks the reference count. If the count is zero or the option is not set, then the routine returns all memory blocks to the system heap, calls each sub-object's and referenced object's kill routine, and adjust the global in-use counter. The reason for an explicitly supplied kill routine is the same as for the free routine.

`static inline void reset_<i>();`

This routine removes and kills every object on the recycle list. If you want to perform additional reset activities, such as resetting buffer limits and freeing working storage, then you should write your own external `Reset_<X>` routine.

`static void write_<i>(<I> *<i>, FILE *output);`
The routine writes an encoding of the object pointed at by `<i>` to the file `output` in a single segment starting at the current cursor position within the file. Upon completion the cursor is at the end of the object's encoding on the file `output`. On may chose to write their own external routine in the event that they wish to write some prologue or epilogue information around the segment written by the core routine.

`static inline <I> *read_<i>(FILE *input);`
The routine reads and generates an object of type `<I>` from file `input` starting at the current cursor position within the file. Upon completion the cursor is at the end of the object's encoding on the file `input`. Any references within such an object are set to `NULL` and it is up to the user to re-establish them if they wish. If a memory block contains information, such as pointers, that are relative as opposed to absolute, then you need to write the external `Read_<X>` routine and after calling the core routine adjust the relative information. Moreover, you may need to read wrapper information on the input file around the input segment read by the core routine in case you did such for the corresponding write routine.

As a smallest possible example, the `XPoint` object of module `xfig.p` has no variable-sized components, no sub-objects, and its record is externally visible. In this case, all the code we needed to write is:

```
MANAGER XPoint
```

Recall from the examples of Convention 4, that a `Tiff` object defined in the `image.p` module is opaque (i.e. declared `void`), but internal to the module it is realized by a `Tio` record that has a `Tiff_Reader` and a `Tiff_Writer` sub-object. All that is required is the `MANAGER` line:

```
MANAGER Tiff(Tio) reader*Tiff_Reader writer*Tiff_Writer
```

As another example, let's consider the `Array` object introduced in Convention 1 which has three variable-sized blocks pointed at by the fields `dims`, `text`, and `data`, where the block pointed at by `data` has a size modeled by a `uint64`. In this case you need to supply a routine `array_<size_name>` for each of the three blocks that returns the size of the given block. Also note the subtle difference that this class has read and write routines as indicated by the flags `-IO` on the `MANAGER` line:

```
static inline uint32 array_nsize(Array *a)
{ return (sizeof(int)*a->ndims); }

static inline uint32 array_dsize(Array *a)
{ return (a->size*type_size[a->type]); }

static inline uint32 array_tsize(Array *a)
{ return (strlen(a->text)+1); }

MANAGER -IO Array dims:nsize data!dsize text:tsize
```

As a final example, consider the `Level_Tree` object from the examples of Convention 4 that is in the module `level.set.p` and that is opaque with internal definition `Comtree`. This object has a vector of `vertex` records pointed at by `level_tree`. The number of elements in the vector is in the field `csize`. The object also has a reference `apart_ref` to the `Array` or `Slice` from which it was made. An interesting aspect of this example is that the `-p` option is set as the pack routine needs to readjust a pointer `regtrees` that is relative to `level_tree` which might be moved by the pack.

```
static inline uint32 comtree_lsize(Comtree *tree)
{ return (tree->csize * sizeof(vertex)); }

MANAGER -pIO Level_Tree(Comtree) level_tree:lsize apart_ref@APart

Level_Tree *Pack_Level_Tree(Level_Tree *etree)
{ Comtree *tree = (Comtree *) etree;
  int pack_failed = pack_comtree(etree);
  etree->regtrees = etree->level_tree - 1;
  if (pack_failed) return (NULL);
  return (etree);
}
```

Hopefully, these examples are enough to get your started. For more examples, simply `grep` the `.p` files of `mylib` for the word `MANAGER` to find all the places where these object manager macros have been used.

3 Memory Management Macro Transformations

Here we will describe in detail the code generated in response to a `MANAGER` declaration. As introduced in the previous section, the notations `<X>`, `<I>`, `<x>`, and `<i>` will denote the external and internal class name in upper and lower case, respectively. In addition,

1. `<block_field>` and `<size_name>` will denote each memory block field and its associated size name
2. `<subo_field>` and `<subo_name>` will denote each sub-object field and its associated class name
3. `<ref_field>` and `<ref_name>` will denote each reference field and its associated class name

The manager generates the same code fragment for every field/name pair and this will be denoted by giving the code for *a* pair and then follow it by ellipses. Occasionally the same code is generated for different kinds of field/name pairs and as an example we would use the template `<(subo|ref)_field>` to denote every sub-object and reference field.

The manager first produces a declaration of the wrapper object `_<I>` (Lines 1-7) that contains an object of type `<I>` (Line 6), fields to hold the sizes of each memory block (Lines 5 repeated), a reference count (Line 4), and link fields `next` and `prev` (Line 2-3) used to implement the recycle list `Free_<I>_List` declared in Line 8 and the inuse list `Use_<I>_List` declared in Line 9. The free list is singly-linked with the `next` field and the inuse list is, in order to support deletion within the list, doubly-linked with both fields.

```
 1. typedef struct __<I>
 2.   { struct __<I>   *next;
 3.     struct __<I>   *prev;
 4.     int             refcnt;
 5.     (uint32|uint64) <size_name>;
        ...
 6.     <I>             <i>;
 7.   } _<I>;

 8. static _<I> *Free_<I>_List = NULL;
 9. static _<I> *Use_<I>_List  = NULL;
```

Then the manager generates a declaration for the global variable `<I>_Offset` (Line 1) that is set up to hold the offset in bytes from the top of a container to the proper object it contains. A global mutex, `<I>_Mutex` (Line 2) is declared and initialized for use in the portions of the code for the object routines that are intrinsically not re-entrant. Specifically any updating of the free and in-use lists and counts as well as changes to the reference count of an object instance. The global variable `<I>_Inuse` (Line 3) holds the number of objects currently in use and is the value returned by `<X>_Usage()` (Lines 4-5). The routines `<X>_Refcount` and `<X>_List` are also generated. `<X>_Refcount` simply returns the current reference count of an object (Lines 6-8). and `<X>_List` walks the forward link of the use list and calls a handler on each object (Lines 9-13).

```
 1. static int   <I>_Offset = sizeof(_<I>) - sizeof(<I>);

 2. static pthread_mutex_t <I>_Mutex = PTHREAD_MUTEX_INITIALIZER;

 3. static int   <I>_Inuse  = 0;

 4. int <X>_Usage()
 5. { return (<I>_Inuse); }

 6. int <X>_Refcount(<X> *<x>)
 7. { _<I> *object = (_<I> *) (((char *) <x>) - <I>_Offset);
 8.   return (object->refcnt);
    }

 9. void <X>_List(void (*handler)(<X> *))
10. { _<I> *a, *b;
11.   for (a = Use_<I>_List; a != NULL; a = b)
12.     { b = a->next;
13.       handler((<X> *) &(a-><i>));
        }
    }
```

For each block field / size name pair a routine `allocate_<i>_<block_field>` is generated that is passed the object with internal type, the requested size of the memory block, and the name of the generator requesting the block (Line 1). It returns a non-zero result if it could not allocate the block. The routine first gets a pointer to the wrapper (Line 2) and if the requested size is larger than the current size of the block (Line 3) then it attempts to expand the block (Line 4). If it cannot (Line 5) then it returns a non-zero value without having made any changes to the object. Otherwise it updates the block pointer and its size field in the wrapper and returns 0 (Line 6-8). In addition, the manager generates the small routine `sizeof_<i>_<block_field>` which given the object returns the padded size of the memory block recorded in the wrapper (Lines 10-12).

```
 1. static inline int allocate_<i>_<block_field>(<I> *<i>, (uint32|uint64) <size_name>,
                                                             char *routine)
 2. { _<I> *object = (_<I> *) (((char *) <i>) - <I>_Offset);
 3.   if (object-><size_name> < <size_name>)
 4.     { void *x = Guarded_Realloc(<i>-><block_field>,<size_name>,routine);
 5.       if (x == NULL) return (1);
 6.       <i>-><block_field> = x;
 7.       object-><size_name> = <size_name>;
        }
 8.   return (0);
    }

10. static inline (uint32|uint64) sizeof_<i>_<block_field>(<I> *<i>)
11. { _<I> *object = (_<I> *) (((char *) <i>) - <I>_Offset);
12.   return (object-><size_name>);
    }
```

The routine `new_<i>` returns a pointer to an object of type `<I>` that is within a wrapper of type `_<I>`, unless it cannot allocate it in which case it returns `NULL`. The size requested for each memory block is passed as a parameter `<size_name>` in order of their declaration to the manager, along with the name `routine` of the generator requesting the object (Line 1). If the recycle list is empty (Line 5) then a wrapper object is `malloc`'d if possible or `NULL` is returned if it is not (Line 6-8). In addition, each memory block has its size set to 0 and its pointer set to `NULL` (Lines 9-10 repeated). If the recycle list is not empty, then the most recently freed object is taken from the list (Lines 11-12). In both cases, a pointer `<i>` is established to the proper object with internal type (Lines 8 and 13). Thereafter, the reference count is set to 1 (Line 14), the number of objects in use is increased by 1 (Line 15), and the object is prepended to the use list (Lines 16-21). Then every sub-object or reference field is set to `NULL` (Line 22 repeated), and each memory block is created or expanded to the desired size (Line 23-25 repeated) if possible, if not the nascent object is killed and `NULL` is returned. A pointer to the object within the wrapper is returned (Line 26). All the code dealing with the free and use lists and reference counts (Lines 5-20) is locked by the object class mutex `<I>_Mutex` in order to make sure that a race-condition does not occur between two threads simultaneously manipulating the same object.

```
 1. static inline <I> *new_<i>((uint32|uint64) <size_name>, ... , char *routine);
 2. { _<I> *object;
 3.    <I>  *<i>;

 4.   pthread_mutex_lock(&<I>_Mutex);
 5.   if (Free_<I>_List == NULL)
 6.     { object = (_<I> *) Guarded_Realloc(NULL,sizeof(_<I>),routine);
 7.       if (object == NULL) return (NULL);
 8.       <i> = &(object-><i>);

 9.       object-><size_name> = 0;
10.       <i>-><block_field> = NULL;
            ...
        }
      else
11.     { object = Free_<I>_List;
12.       Free_<I>_List = object->next;
13.       <i> = &(object-><i>);
        }
14.   object->refcnt = 1;
15.   <I>_Inuse += 1;
16.   if (Use_<I>_List != NULL)
17.     Use_<I>_List->prev = object;
18.   object->next = Use_<I>_List;
19.   object->prev = NULL;
20.   Use_<I>_List = object;
21.   pthread_mutex_unlock(&<I>_Mutex);

22.   <i>-><(subo|ref)_field> = NULL:
        ...
23.   if (allocate_<i>_<block_field>(<i>,<size_name>,routine))
24.     { kill_<i>(<i>);
25.       return (NULL);
        }
        ...

26.   return (<i>);
    }
```

The routine `Inc_<x>` simply accesses the wrapper (Line 2), increments its reference count by 1 (Line 4), and then returns a pointer to the object (Line 6), where the increment is protected by the object class mutex lock (Lines 3 and 5).

```
 1. <X> *Inc_<X>(<X> *<x>)
 2. { _<I> *object = (_<I> *) (((char *) <x>) - <I>_Offset);
 3.   pthread_mutex_lock(&<I>_Mutex);
 4.   object->refcnt += 1;
 5.   pthread_mutex_unlock(&<I>_Mutex);
 6.   return (<x>);
    }
```

If a flag for a core routine, e.g. `-c` for `copy_<i>`, is not set, then code for the external routine, e.g. `Copy_<X>`, is generated wherein the routine simply calls the core routine! For example,

```
    <X> *Copy_<X>(<X> *<x>)
    { return ((<X> *) copy_<i>((<I> *) <x>)); }
```

Thus the core routine is always generated except for the read and write routines when no flag for them is set. So from here on out we describe just the code for the core routines.

The core routine `copy_<i>` is generated as shown below. First a new object is requested from `new_<i>` where the size of each memory block is requested to be its size in the object `<i>` being copied (obtained by calling the user-supplied routine `<i>_<size_name>`) (Line 2). Then a copy, named `_<block_field>`, of each block field pointer of the copy is saved (Line 3 repeated) before assigning the structure of `<i>` to `copy` in order to set all scalar values of the record (Line 4). Then each memory block field pointer is restored and the value of the block copied if it is not `NULL` (Lines 5-7 repeated). Also every sub-object field is copied if the pointer is not `NULL` (Lines 8-9 repeated), and every reference field is incremented if not `NULL` (Lines 10-11 repeated). The copy is returned as the result (Line 12).

```
 1. static inline <I> *copy_<i>(<I> *<i>)
 2. { <I> *copy = new_<i>(<i>_<size_name>(<i>), ... <i>_<size_name_N>(<i>),"Copy_<X>");

 3.   void *_<block_field> = copy-><block_field>;
      ...

 4.   *copy = *<i>;

 5.   copy-><block_field> = _<block_field>;
 6.   if (<i>-><block->field> != NULL)
 7.     memcpy(_<block_field>,<i>-><block_field>,<i>_<size_name>(<i>));
      ...

 8.   if (<i>-><subo_field> != NULL)
 9.     copy-><(subo_field> = Copy_<subo_name>(<i>-><subo_field>);
      ...

10.   if (<i>-><ref_field> != NULL)
11.     Inc_<ref_name>(<i>-><ref_field>);
      ...

12.   return (copy);
    }
```

The core routine `void pack_<i>` first sets a pointer to the wrapper containing `<i>` (Line 2). Then for each memory block, it checks if the block is bigger than the value within it (Line 3) and if so, the unpadded size is recorded in the variable `ns` (Line 4). If `ns` is non-zero then the memory block is shrunk if possible, if not a non-zero value is returned (Lines 6-8). If the unpadded size is 0 then the block is freed and its pointer set to `NULL` (Lines 9-10). Then the new size of the block is set (Line 11). For every sub-object, the associated pack routine is called if the sub-object reference is not `NULL` and if the pack fails then a non-zero value is returned (Lines 12-13 repeated). If all succeeds then zero is returned (Line 14).

```
 1. inline int pack_<i>(<I> *<i>)
 2. { _<I> *object = (_<I> *) (((char *) <i>) - <I>_Offset);

 3.   if (object-><size_name> > <i>_<size_name>(<i>))
 4.     { uint64 ns = <i>_<size_name>(<i>);
 5.       if (ns != 0)
 6.         { void *x = Guarded_Realloc(<i>-><block_field>,ns,"Pack_<X>");
 7.           if (x == NULL) return (1);
 8.           <i>-><block_field> = x;
          }
        else
 9.         { free(<i>-><block_field>);
10.           <i>-><block_field> = NULL;
          }
11.       object-><size_name> = ns;
      }
      ...

12.   if (<i>-><subo_field> != NULL)
13.     if (Pack_<subo_name>(<i>-><subo_field>) == NULL) return (1);
      ...
14.   return (0);
    }
```

The core routine `free_<i>` first gets a point to the object's wrapper (Line 2). Then the object class mutex is locked (Line 3), the reference count is decremented and if it is non-zero thereafter, the routine simply returns (Lines 4-6) being careful to unlock the mutex. The routine also checks that the reference count is not negative indicating this object is already free or killed (Lines 7-8).

Otherwise the routine proceeds with freeing the object by removing it from the use list (Lines 9-14), placing it on the recycle list (Lines 15-16), decrementing the number of objects in use (Line 17), and finally unlocking the object class mutex (Line 18). It finishes by freeing every non-null pointer to a sub-object or reference (Lines 19-21 repeated),

```
 1. void free_<i>(<I> *<i>)
 2. { _<I> *object = (_<I> *) (((char *) <i>) - <I>_Offset);

 3.   pthread_mutex_lock(&<I>_Mutex);
 4.   if (--object->refcnt > 0)
 5.     { pthread_mutex_unlock(&<I>_Mutex);
 6.       return;
        }

 7.   if (object->refcnt < 0)
 8.     fprintf(stderr,"Warning: Freeing previously released Array\n");

 9.   if (object->prev != NULL)
10.     object->prev->next = object->next;
11.   else
12.     Use_<I>_List = object->next;
13.   if (object->next != NULL)
14.     object->next->prev = object->prev;

15.   object->next = Free_<I>_List;
16.   Free_<I>_List = object;
17.   <I>_Inuse -= 1;
18.   pthread_mutex_unlock(&<I>_Mutex);

19.   if (<i>-><(subo|ref)_field> != NULL)
20.     { Free_<(subo|ref)_name>(<i>-><(subo|ref)_field>);
21.       <i>-><(subo|ref)_field> = NULL;
        }
      ...

    }
```

The core routine `kill_<i>` first gets a point to the object's wrapper (Line 2). Then the object class mutex is locked (Line 3), the reference count is decremented and if it is non-zero thereafter, the routine simply returns (Lines 4-6) being careful to unlock the mutex. The routine also checks that the reference count is not negative indicating this object is already free or killed (Lines 7-8). Otherwise the routine proceeds with killing the object by removing it from the use list (Lines 9-14), and decrementing the number of objects in use (Line 15). At this point the mutex can be unlocked (Line 16) and the routine finishes by killing every non-null pointer to a sub-object or reference (Lines 17-18 repeated), returning every memory bock to the system heap (Lines 19-20 repeated), returning the object's wrapper to the system heap (Line 21).

```
 1. void kill_<i>(<I> *<i>)
 2. { _<I> *object = (_<I> *) (((char *) <i>) - <I>_Offset);

 3.   pthread_mutex_lock(&<I>_Mutex);
 4.   if (--object->refcnt > 0)
 5.     { pthread_mutex_unlock(&<I>_Mutex);
 6.       return;
        }

 7.   if (object->refcnt < 0)
 8.     fprintf(stderr,"Warning: Freeing previously released Array\n");

 9.   if (object->prev != NULL)
10.     object->prev->next = object->next;
11.   else
12.     Use_<I>_List = object->next;
13.   if (object->next != NULL)
14.     object->next->prev = object->prev;
15.   <I>_Inuse -= 1;
16.   pthread_mutex_lock(&<I>_Mutex);

17.   if (<i>-><(subo|ref)_field> != NULL)
18.     Kill_<(subo|ref)_name>(<i>-><(subo|ref)_field>);
      ...

19.   if (object-><size_name> != 0)
20.     free(<i>-><block_field>);
      ...

21.   free(((char *) <i>) - <I>_Offset);
    }
```

The core reset routine `reset_<i>` removes every object from the recycle list and kills it (Lines 5-11). Each object is removed from the free list in Lines 6-8 and killed in Lines 9-11, by returning all of its memory blocks and the object container to the heap. Note carefully that the code is locked by the object class mutex (Lines 4 and 12).

```
 1. void reset_<i>()
 2. { _<I> *object;
 3.   <I>  *<i>;
 4.   pthread_mutex_lock(&<I>_Mutex);
 5.   while (Free_<I>_List != NULL)
 6.     { object = Free_<I>_List;
 7.       Free_<I>_List = object->next;
 8.       <i> = &(object-><i>);

 9.       if (object-><size_name> != 0)
10.         free(<i>-><block_field>);
          ...

11.       free(object);
        }
12.   pthread_mutex_lock(&<I>_Mutex);
    }
```

The core write routine `write_<i>` first writes the string `"<X>"` to the output (Line 2) and then writes the structure of the object (Line 3). Then it writes each memory block (Lines 4-5) and sub-object (Lines 6-7) in the order in which they are declared in the `MANAGER` specification line.

```
 1. void write_<i>(<I> *<i>, FILE *output)
 2. { fwrite("<X>",<len(X)>,1,output);
 3.   fwrite(<i>,sizeof(<I>),1,output);

 4.   if (<i>_<size_name>(<i>) != 0)
 5.     fwrite(<i>-><block_field>,<i>_<size_name>(<i>),1,output);
      ...

 6.   if (<i>-><subo_field> != NULL)
 7.     Write_<subo_name>(<i>-><subo_field>,output);
      ...
    }
```

The core read routine `read_<i>` first reads a string of length equal to the external class name and verifies that it is indeed the class name, returning `NULL` if it is not (Lines 2-5). Then a new object is created with all memory blocks initially unallocated (Line 6), `NULL` being returned if this fails. Then a copy of the object record is made in `read` and each sub-object field is set to `NULL` so that the kill in Line 20 works if the read needs to fail in what follows (Lines 8-9). Next the top level structure is read (Line 10). Then it allocates and reads each memory block of non-zero size (Lines 11-14), reads each non-`NULL` sub-object (Line 15-17), and sets each reference field to `NULL` (Line 18) in the order in which they are declared in the `MANAGER` specification line. All the allocations, sub-object reads, and file reads are checked and if any fail, the code jumps to `error`, the object is killed, and `NULL` is returned (Lines 20-22).

```
 1. <I> *read_<i>(FILE *input)
 2. { char name[<len(X>)];
 3.   fread(name,<len(X)>,1,input);
 4.   if (strncmp(name,"<X>",<len(X)>) != 0)
 5.     return (NULL);

 6.   <I> *obj = new_<i>(0 ... ,"Read_<X>");
 7.   if (obj == NULL) return (NULL);

 8.   <I> read = *obj;

 9.   obj-><subo_field> = NULL;
      ...

10.   if (fread(obj,sizeof(<I>),1,input) == 0) goto error;

11.   <i>-><block_field> = NULL;
12.   if (<i>-><size_name>(obj) != 0)
13.     { if (allocate_<i>_<block_field>(obj,<i>_<size_name>(obj),"Read_<X>")) goto error;
14.       if (fread(obj-><block_field>,<i>_<size_name(obj),1,input) == 0) goto error;
        }
      ...

15.   if (read.<subo_field> != NULL)
16.     { obj-><subo_field> = Read_<subo_name>(input);
17.       if (obj-><subo_field> == NULL) goto error;
        }
      ...
```

```
  obj-><ref_field> = NULL;
  ...

  return (obj);

error:
  kill_<i>(<I>);
  return (NULL);
}
```