

# Customising Object Allocation

Giuseppe Attardi and Tito Flagella \*

Dipartimento di Informatica, Università di Pisa  
Corso Italia 40, I-56125 Pisa, Italy  
net: {attardi,tito}@di.unipi.it

**Abstract.** Automatic garbage collection relieves programmers from the burden of managing memory themselves and several techniques have been developed that make garbage collection feasible in many situations, including real time applications or within traditional programming languages. However optimal performance cannot always be achieved by a uniform general purpose solution. Sometimes an algorithm exhibits a predictable pattern of memory usage that could be better handled specifically, delaying as much as possible the intervention of the general purpose collector. This leads to the requirement for algorithm specific customisation of the collector strategies. We present a dynamic memory management framework which can be customised to the needs of an algorithm, while preserving the convenience of automatic collection in the normal case. The Customisable Memory Management (CMM) organizes memory in multiple heaps. Each heap is defined as a C++ class which encapsulates a particular storage discipline. The default heap for collectable objects uses the technique of mostly copying garbage collection, providing good performance and memory compaction. Customisation of the collector is achieved exploiting object orientation by defining specialised versions of the collector methods for each heap class. The object oriented interface to the collector enables coexistence and coordination among the various collector as well as integration with traditional code unaware of garbage collection. The CMM is implemented in C++ without any special support in the language or the compiler. The techniques used in the CMM are general enough to be applicable also to other languages.

## 1 Introduction

In most programming languages, memory allocation is either under total responsibility of the programmer or under full control of a garbage collector.

The garbage collector's function is to find data objects that are no longer in use and to reclaim their space for further use by the program. An object is considered *garbage*, and therefore subject to reclamation, if it is not reachable

---

\* The research described here has been funded in part by the ESPRIT Basic Research Action, project PoSSo.

Part of this work has been done while the first author was visiting the International Computer Science Institute, Berkeley, California.

by the program via any path of pointer traversal. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never follow a “dangling pointer” leading to a deallocated object.

This technique has several advantages since it improves: *safety*, avoiding the risk of deallocating an object too soon; *accuracy*, avoiding the risk of forgetting to deallocate unused memory; *simplicity*, assuming a computational model with unlimited memory; *modularity*, the program does not have to be interspersed with bookkeeping code not related to the application; *burden* on programmers who are relieved from taking care of memory management.

Garbage collection has been mostly available in programming languages whose design had taken into account its requirements: for instance not allowing pointer manipulation (Lisp), using tagged pointers or providing run-time type information, using special notation for pointer operations (Simula), requiring enhanced pointer declarations (Modula3).

This has restricted garbage collection from general use, more than the often cited concerns about efficiency. Recent research has proved in fact that the performance of garbage collectors compares quite well with explicit memory deallocation (using primitives like `free` or `delete`) [17], and techniques like *generational* garbage collection have been developed to minimise latency during garbage collection.

With the development of techniques for *conservative* garbage collection, the use of garbage collection has become feasible also for languages which are not well behaved with respect to pointers. Even so, current implementations limit the use of pointers: for instance in Modula3 [11] there are traced pointers to collected objects and untraced pointers to uncollected objects – an uncollected object can’t contain an untraced pointer.

One problem still needs to be addressed: interacting with the collector, which in general assumes full control of memory management.

The fact that a collector assumes total control of memory management can sometimes be a drawback. For instance, it becomes harder to integrate code or libraries which are unaware of garbage collection and use pointers without restrictions, it is impossible to mix code from programming languages with different memory models. And finally, as we argue here, it is impossible to specialise the collector to the particular needs of an algorithm.

A general purpose collector strategy may work well in most circumstances, but there are cases where an algorithm within an application exhibits a predictable pattern of memory usage which can be exploited to achieve significant performance benefits. When the collector is alone in control of memory management, it becomes impossible to arrange for allocating and deallocating objects in a special way during the execution of that portion of the application. The programmer could request an area of memory from the collector and arrange for managing it by himself. This however would not be sufficient if pointers are allowed from within such area to objects external to it. Such objects might still be reachable but the general collector would not be aware of them and might unduly reclaim their space. Therefore if the user wants to manage memory by

himself, some form of coordination with the general collector is necessary.

We faced a situation like this when developing memory management facilities for a large research project in symbolic algebra: the ESPRIT BRA PoSSo which aims at building a sophisticated system for solving systems of polynomial equations. The core algorithm of PoSSo is quite memory intensive and even the best traditional garbage collection techniques lead to thrashing where most of the time is spent in garbage collection. However, there are precise points in the algorithm where all data created during the previous step of the algorithm become irrelevant and can be deallocated in block. By customising the allocation within this portion of the algorithm significant improvements in performance have been achieved.

The requirements of this project led us to design a Customisable Memory Management (CMM) framework where several policies can coexist. Users can choose the most appropriate one, ranging from manual management to fully automatic garbage collection, and can also implement their own specialised memory management. The extensibility of the framework is achieved exploiting the object oriented paradigm of C++, thereby maintaining a consistent and simple interface for programmers.

The CMM consists of:

1. a general purpose garbage collector for C++; this collector is called *primary garbage collector* and is a variant of Bartlett's mostly copying collector [4];
2. a user interface: the interface used by programmers to access the CMM;
3. a programmer interface: a set of facilities used by CMM programmers to define specific memory management policies as appropriate for their applications.

In the rest of the paper, we introduce the idea of custom object allocation, the requirements for a customisable memory manager. After recalling the general principles of memory management, we present our primary collection algorithm, then discuss the CMM, its implementation and its usage. Finally we illustrate how to emulate different garbage collector styles and application specific memory management policies.

## 2 Custom Object Allocation

Among the properties of the storage for an object that one would like to be able to control are:

- lifetime
- relocatability
- traversability

For instance, if an object contains some implicit pointers to within itself, like a branch instruction in a binary code segment, one needs to specify that the object cannot be relocated. Supplying information about the layout of an object

may be useful to improve the accuracy and the performance of the garbage collector. For instance, an array of characters need not be traversed looking for pointers to other objects. Specifying the lifetime of an object is more difficult, but there are some useful simple cases: for instance asserting that an object is permanent may be useful to store it in an area which is visited less frequently by the collector; objects with a dynamic extent can be allocated on the stack.

In most garbage collector implementations, such properties are dictated by the collector design and users have no control on them: if the collector is a copying collector, it will not handle objects which cannot be moved. The lifetime of dynamically allocated objects is unpredicable and uncontrollable.

To provide user control over these properties entails adding a new dimension to garbage collection: customisation of object allocation, applicable to individual objects rather than to the whole collector or to classes of objects. Customisation requires a collector designed to be open and to delegate portions of his task to other collectors.

This is a different concept from the mechanisms of parametrisation or tuning that some collectors provide.

For instance garbage collection intervention can be sometimes avoided in ADA [1] by specifying an upper bound for the space needed for data of a certain type. The corresponding space can then be reserved globally when the definition is elaborated. Subsequently, when leaving the program unit enclosing the type definition, the space corresponding to the collection may be recovered since the contained objects are no longer accessible.

An interesting form of tuning is provided in Lisp Machine Lisp [14] where one can define areas of memory and designate which area to use when allocating objects. Areas are primarily used to give the user control over the paging behaviour of a program. One area could be selected as permanent, so that it would not be copied at each activation of the ephemeral collector. Microcode support was present in the Lisp Machine so that each area could potentially have a different storage discipline, but apparently such feature was never exploited.

Information about traversal of objects can be supplied to the Boehm-Weiser collector for C [5] in the form of a region parameter to the allocation routine. Region identification is used to determine how to locate pointers within objects during traversal by the collector. The `PTRFREE` region for instance is used to allocate objects which do not contain pointers. Such regions are simply skipped by the collector. Detailed traversal information for each type of object is instead required in Bartlett's mostly-copying collector [4].

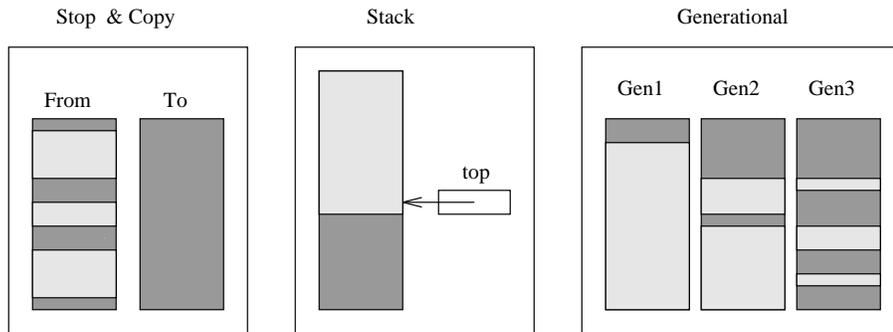
In all these examples however the collector implements a fixed policy, and no alternative is contemplated. The collector routines at most take into account the area where an object resides besides its type and layout.

The CMM allows users to customise object allocation by specifying individually for each object created which policy to adopt for its storage. The CMM admits the presence of several collectors, each one in charge of its own heap, which coordinate with each other for proper memory management. The heap where an object resides determines the policy used for the object, but to achieve

coordination it is not enough for the procedures of the collector to discriminate on the heap of residence, they must also take into account which heap is currently subject to collection. This two dimensional dependency is an original feature of the CMM and it will be discussed later.

CMM users can select among a few predefined memory management disciplines, define their own, or customise those provided in the framework exploiting the mechanisms of inheritance and specialization.

For instance it is conceivable a situation like in the following figure, where three different memory management policies are available or even used together in the same application: a traditional stop-and-copy collector, a specialised stack allocator for portions of the algorithm with controlled behaviour and a generational collector for real-time tasks such as user interfaces.



The mechanism to implement these alternative policies is the *heap* abstraction which we develop in this paper. Specific algorithms are used and particular data structures are maintained by each heap to ensure its proper behaviour. A critical question is what to do with pointers which cross the boundaries of heaps. If no such pointers are allowed, then a heap need only be concerned with objects it has allocated and over which it has some control. We considered this solution too restrictive, since it would not allow portions of applications built separately by different people to exchange data freely. We took therefore special care to design the mechanism of heaps to ensure that different heaps can coexist and data of different sources can be mixed. The amount of coordination necessary to achieve this goal consists of a traversal function that each class of collectable objects must provide and a scavenging function for each heap. To achieve coordination in a simple and effective way, we exploit the object oriented features of C++. In practice, all the operations of the collector are performed through member functions of the class of each object. However, the action of the collector on an object may also depend on the heap where the collection started, not just on the heap to which the object belongs. For instance if the collection starts in the Stop&Copy heap, it applies its methods to mark and traverse the object in that heap, but if a pointer leads into a Stack heap, those objects are unobtrusively traversed without modifying them. Only if such traversal leads back into the original heap, will the full collector operation resume.

### 3 Design Issues

In designing the CMM we tried to achieve the following goals:

- *portability*: the CMM is simply a library of C procedures and C++ classes, which can be used with any C++ compiler. Alternative solutions rely on changes to the underlying language or compiler.
- *coexistence*: code and objects built with the CMM can be exchanged with traditional code and libraries. No restrictions exist on whether a collected object can point to a non collected object and viceversa. We wanted to be able to pass collected objects to programs unaware of garbage collection, allowing them to store such objects in data structures, without special burden on the programmer or risk that the object would be garbage collected. Alternative solutions require the programmer to put an object in an “escape list” before passing it to an external procedure.
- *algorithm specific customisation*: the allocation policy can be customised to the particular needs of an algorithm. This is different from other solutions, where the allocation policy is associated to the type of an object [10]. For the purpose of our applications, it is necessary to allocate the same type of object sometimes with one policy and sometimes with another. For example, in PoSSo there is only one class of polynomials, but sometimes a polynomial is allocated in a special heap which can be freed quickly once a certain portion of the simplification algorithm is complete; in other cases the lifetime of the polynomial cannot be predicted, so it must be allocated in the general heap.
- *multiple logical heaps*: at least two heaps are necessary, one for collectable objects and one for traditional objects. However two is not enough: for instance collectable objects containing data which cannot be relocated for some reasons must be handled differently from other objects which are copied by the collector. For this reason the CMM provides multiple logical heaps.
- *usability*: only a minimal burden is placed on the programmer who wants to use the collector. When collectable objects are required the programmer needs to define their class as inheriting from the base class `GcObject` and supply a method for traversing them, a task which could be automated.
- *separation of concerns*: memory management code needs not to be included within algorithms, and it is possible to change the memory policy just by selecting which heap is employed by the algorithm.
- *efficiency*: the implementation is efficient enough to be as good as and sometimes better than hand tuned allocation.

The CMM allows customisation of the collector and provides a few pre-built variants. One could argue whether a single general strategy could fit all the needs. For instance a generational garbage collector ensures that memory is reclaimed quickly. However not even a generational garbage collection is good enough for applications like PoSSo where one must prevent or delay garbage collection as much as possible, not just make its duration shorter. For the vast majority of applications a general purpose strategy is adequate, and the CMM provides a

good one by default. But for research or applications that need to push the limits of technology, the CMM provides a solution with limited burden on the user.

## 4 Dynamic Memory Management: Concepts and Terminology

A garbage collection mechanism basically consists of two parts [16]:

1. distinguishing the *live objects* from the garbage in some way, or *garbage detection*;
2. reclaiming the garbage objects' storage, so that the running program can reuse it.

The formal *criterion* to identify *live* objects is expressed in terms of a *root set* and *reachability* from these roots. The *root set* consists of the global and local variables, and any registers used by active procedures. Heap objects directly reachable from any of these variables can potentially be accessed by the running program, so they are *live* objects which must be preserved. In addition, since the program might traverse pointers from these objects to reach other objects, any object reachable from a live object is also live. Thus the *live set* is the set of objects in some way reachable from the roots. Any object not in the live set is garbage and can be safely reclaimed.

Depending on the kind of information available during the traversal of objects from the root set, a tracing collector can be *conservative*, *type-accurate* or both.

A *conservative* garbage collector does not require cooperation from the compiler and assumes that anything that *might* be a pointer actually *is* a pointer. In this case an integer (or any other value) is assumed to be a pointer by the collector if it corresponds to an address inside the current heap range: any such value is called an *ambiguous pointer*. A root containing an ambiguous pointer is called an *ambiguous root*. A garbage collector is *type-accurate* when it is able to distinguish which values are genuine pointers to objects. Some garbage collectors adopt a combination of these two techniques: some pointers are dealt conservatively, while others are treated in a type accurate way.

The main limitations of a purely conservative collector are memory fragmentation in applications dealing with objects of several sizes, which arises from the inability to move objects, and the risk that a significant amount of memory might not be reclaimed in applications with densely populated address spaces of strongly connected objects [15].

The alternative approach which is *type-accurate* in identifying objects faces some problems with hidden pointers. For instance in C++ the location on the stack of the pointer to the object itself, denoted by the variable `this`, is only known to the compiler. The only compiler-independent way to catch such pointers is to examine the stack conservatively. Failing to trace hidden pointers may lead to *dangling pointers* and produce serious consequences for the integrity of the program.

Both these limitations are avoided in the partially conservative approach proposed by Bartlett for his *mostly copying garbage collector*. We chose this technique as the basis for developing our customisable collector.

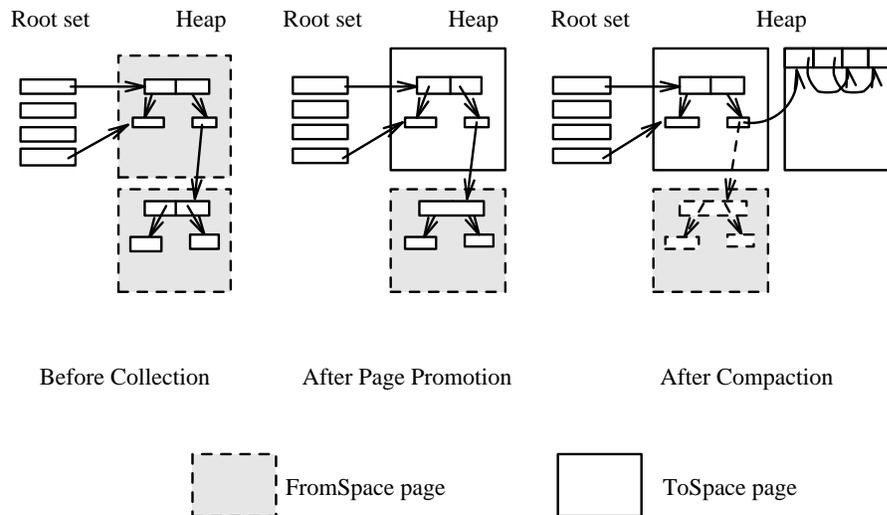
## 5 The Primary Collector

The CMM relies on an underlying general mechanism for identifying objects, moving them and recovering memory. These mechanisms constitute the *primary collector* of the CMM and are based on Bartlett's technique [3]. The difference and the derivation of our technique from Bartlett's original are discussed in [2]. Here we present our implementation.

### 5.1 CMM mostly copying collector

A mostly-copying garbage collector performs compacting collection in the presence of ambiguous pointers in the root set. The technique is an evolution of the classical stop-and-copy collector which combines copying and *conservative* collection. Those objects which are referenced by ambiguous roots are not copied, while most other live objects are copied.

The heap used by the mostly-copying collector consists of a number of equal size pages, each with its own *space-identifier* (either *From* or *To* in the simplest non generational version). The *FromSpace* consists of all pages whose identifier is *From*, and similarly for *ToSpace*. The collector conservatively scans the stack and global variables looking for potential pointers. Objects referenced by ambiguous roots are not copied, while most other live objects are copied. If an object is referenced from a root, it must be scavenged to survive collection.



Since the object cannot be moved, the whole page to which it belongs is saved. This is done by *promoting* the page into *ToSpace* by simply changing its page space-identifier to *To*. At the end of this promoting phase, all objects belonging to pages in *FromSpace* can be copied and compacted into new pages belonging to *ToSpace*. Root reachable objects are traversed with the help of information provided by the application programmer: the programmer must supply the definition for a member function for each class of objects which traces the internal pointers within objects of that class.

Our algorithm uses a bit table called **LiveMap**, to identify objects reached during traversal, improving both virtual memory performance and ability to reclaim storage with respect to Bartlett's algorithm.

The algorithm of the collector is as follows:

1. Clear the **LiveMap** bitmap
2. Scan the root set to determine objects which cannot be moved. Any directly reachable object is marked as *live* setting a bit in the **LiveMap** bitmap and the page to which it belongs is promoted.
3. Scan each promoted page linearly, looking for live objects. Traverse each live object by applying the following procedure to each pointer it contains:
  - (a) if the pointer lays outside the heap do nothing;
  - (b) if it points to an object not yet reached: scavenge the object if it belongs to a non promoted page, i.e. copy it, mark the copy as *live*, set a forwarding pointer within the object to the copy. Otherwise mark the object *live* and, in case it is past the current scanning position, recursively traverse it.
  - (c) if it points to a *live* object in a non promoted page update the pointer to the forward position.

All new pages allocated for copying reachable objects belong to *ToSpace*, therefore the algorithm does not need to recursively traverse copied objects. A copied object is traversed when the collector examines its page, so traversal is rarely recursive.

In this algorithm we have reduced the amount of overhead required in each object to just whatever C++ needs for implementing classes with virtual functions, eliminating the header used in Bartlett's implementation which contained a forward bit, the size of the object and the identifier of a callback routine.

## 6 Multiple Heaps

Besides the copy-collected heap, also the traditional uncollected heap must still be supported by providing the primitives `malloc` or `new` on uncollected classes. The uncollected heap cannot be eliminated since there are programs and libraries which may use uncollected objects in an unsafe way for the collector [10], and there are objects that can't be relocated. However, we must allow objects in the uncollected heap to point to objects in the collected heap and viceversa.

The collector algorithm described previously relies on the fact that all locations which might contain ambiguous pointers are known in advance: they coincide with the root set. Therefore pages to be promoted can be identified by a single linear scan of the root set, in the promotion step of the algorithm.

However, when multiple heaps are present and pointers across heaps are allowed, ambiguous pointers might be detected at a later stage.

Requiring that such pointers be registered as roots is not practical, since it would entail registering as root any collected object which is passed to an external library procedure, which might store such pointer internally. This can be cumbersome to do and may be accidentally forgotten.

Modifying the promotion step of the algorithm to perform a complete traversal from the root set in order to identify ambiguous pointers to collected objects, would be a costly solution with live objects being traversed twice.

In [2] we propose a slight variation to our basic algorithm. We do not update pointers immediately when an object is copied, but we just record the location to be updated, using a temporary bitmap. If we discover later that the object should not have been moved, but rather the page should have been promoted, we restore all the objects in such page from their copies. The updates to pointers are performed only at the end of the algorithm, using the bitmap and the forwarding pointer stored in the objects. This technique is similar to the one suggested by Detlefs [7] to handle C/C++ unions of pointers and non-pointers.

## 6.1 User Collected Heaps

With the algorithm described so far, two heaps are available: an uncollected heap for non garbage collected objects and a collected heap.

Our goal is to allow users to build their own heaps with specific allocation strategies for their applications.

We must however fulfill some essential requirements for the solution to be consistent and practical:

- allow pointers across heaps: restricting the range of pointers is difficult and inconvenient.
- transitivity of liveness: if an object is pointed to by a live object it is live as well. We must ensure that a pointer crossing heap boundaries does not go unnoticed by the collector.
- independence of collectors: it must be possible to write a collector for a particular heap, without relying on the collectors for other heaps, provided the root set for this heap is known.
- coordination among heaps: a simple set of conventions is established to ensure that pointers across heaps can be properly traversed.

In the following figure three heaps are present: the uncollected, the copy collected, and one user collected heap.



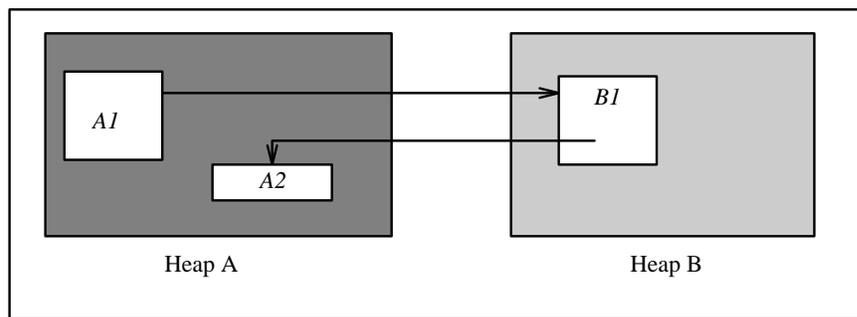
### 6.3 Coordination

To achieve coordination among collectors for the various heaps, one has to agree to a mechanism that allows traversing objects in different heaps on behalf of the collector for another heap. While traversing a foreign heap, a collector should not be allowed to make changes to the objects it visits, except to update recognised pointers to an object in its own heap, after the object has been moved.

This means that one must perform scavenging only for objects in the heap being collected. In other words the scavenge procedure must remain the same throughout a collection, but the scavenge for one heap must not operate on objects in other heaps. **scavenge** is then implemented a member function of each heap class.

**traverse** instead must be specialised according to the type of the object, so we implement it as a member function of each class of objects.

The following figure illustrates the interplay between **scavenge** and **traverse**:



Suppose a garbage collection is started in heap *A* which uses a copy collector. While traversing object *A1*, the garbage collector identifies a pointer to the object *B1*, belonging to heap *B*. Object *B1* is scavenged by the **scavenge** function of the heap *A*. This function recognizes object *B1* as external to heap *A*, so it does not copy the object, as it would if it were internal to the heap, but only traverses the object to determine whether further objects in heap *A* can be reached from it. The behaviour of **scavenge** changes again when object *A2* is reached which belongs to heap *A*. Applying the scavenge function of heap *A* has the effect of copying object *A2*.

## 7 The CMM Run Time

Heap memory is divided into pages of equal size. The allocator for each **Heap** requests pages from the low level page allocator, where to allocate its objects. Each page is tagged with the heap to which it belongs.

The CMM provides a **malloc** routine which uses such pages to allocate objects, implementing the traditional uncollected heap. **malloc** actually creates an instance of class **CmmObject**, which contains an array of the required size, and

returns a pointer to one such array, as expected by calling programs. This is in fact an *interior pointer* inside an object, and we exploit the ability of the CMM to map such pointers to their base. This allows us to traverse also `CmmObjects` by means of its member function `traverse`, defined as follows:

```
void CmmObject::traverse() {
    for (int i = 0; i < size(); i++)
        promote_page(block[i]);
}
```

so that it promotes pages which are pointed to from within the block. The only essential information that `CmmObject` must provide is the size of the block.

In all other collected heaps, the objects allocated are instances of the class `GcObject` or its derivatives, which have their specialised version of `traverse`. No space overhead is present in `GcObject` except for what C++ must supply for the support of virtual functions.

A bitmap is used to deal with internal pointers to objects. Whenever a CMM object is created, the bit corresponding to its first word is set. Using this information, a pointer inside that object can be normalized to the beginning of the object, simply scanning the bitmap backward until the first set bit is found.

When an object has been moved, its first word is replaced by a forwarding pointer to the new object. As already mentioned, this happens only during garbage collection and the collector can determine this situation from the fact that the object is marked *live* and it is in a page in *FromSpace*.

## 7.1 The `GcObject` class

The run time support required for collectable objects is provided by the class `GcObject`. Every class of collectable objects must be derived from `GcObject`.

Users access the services of the CMM mainly by using `GcObject` member functions. The most notable function of `GcObject` is the overloaded `new` operator which takes care of allocating the object in a specific heap. The other functions are used by the primary collector or by user defined collectors.

Here is the public interface for this class.

```
class GcObject
{
public:

    void* operator new(size_t, Heap* = (Heap *)heap);
    virtual void traverse();

    GcObject *next();           // returns the next adjacent object
    bool forwarded();          // tells whether the object has
                                // been forwarded
    void SetForward(GcObject *ptr); // sets the forwarding pointer
}
```

```

GcObject *GetForward();    // returns the forward location
                          //   of the object
Heap *heap();              // returns the heap to which the
                          //   object belongs
void mark();               // marking primitives
bool IsMarked();
void SetLiveMap();
}

```

## 8 CMM User Interface

A collected class must be derived from the class `GcObject` provided by the CMM. The default collector calls the method `traverse` on collected objects to identify their internal pointers to other objects. Users have to provide `traverse` methods for each class whose data members contain pointers. `traverse` must be defined according to well defined rules presented below, because it implements the interface between the CMM and user defined collected objects.

These rules ensure that superclasses or class objects contained in the class are correctly handled. The following example illustrates the rules, which are a generalisation of those in [4]. Suppose the following collected classes were defined:

```

class BigNum: public GcObject
{
    long data;
    BigNum *next;           // Rule (a) applies here
    void traverse();
}

class monomial: private BigNum    // Rule (c) applies here
{
    PowerProduct pp;          // Rule (b) applies here
    void traverse();
}

```

A `BigNum` stores in `next` a pointer to a collected object which needs to be scavenged, so `traverse` becomes:

```

void BigNum::traverse()
{
    scavenge(&next);        // Applying rule (a)
}

```

Because `monomial` inherits from `BigNum`, the method `traverse` for this base class must be invoked; finally, since a `monomial` contains a `BigNum` in `pp`, this object must be traversed as well:

```

void monomial::traverse()
{
    BigNum::traverse();           // Applying rule (c)
    pp.traverse();               // Applying rule (b)
}

```

Finally, to deal with multiple base classes, we must identify the hidden pointer to the base class present inside an object. This cannot be done in a compiler independent way, so the CMM provides a macro `VirtualBase` which is compiler specific. For instance, its definition for the GNU C++ compiler is:

```
#define VirtualBase(A) & (_vb$ # A)
```

In summary the rules are:

- (a) for a class containing a pointer, say `class C { type *x; }`, the method `C::traverse` must contain `scavenge(&x)`
- (b) for a class containing an instance of a collected object, say `class C { GcClass x; }`, the method `C::traverse` must contain `x.traverse()`
- (c) for a class derived from another collected class, say `class C:GcClass {...}`, the method `C::traverse` must contain `GcClass::traverse()`.
- (d) for a class deriving from a virtual base class, say `class C: virtual GcClass {...}`, the method `C::traverse` must contain `scavenge(VirtualBase(GcClass))`;

Preprocessing [8] or compiler support [13] could be adopted to avoid hand coding of these functions and risks of subtle errors in programs. We plan to address this issue in the future.

## 8.1 Object Creation

When creating a collected object one can specify in which heap to allocate it. The parameter `heap` can be supplied in the standard C++ placement syntax for the `new` operator:

```
p = new(heap) Person(name, age);
```

If the user does not specify any heap, the default heap `heap` is used:

```
p = new Person(name, age);
```

which is equivalent to:

```
p = new(heap) Person(name, age);
```

where `heap` is a global variable initialised to the system heap.

When creating collected objects, the programmer can decide case by case where to allocate them. In summary, the following are the alternatives for object allocation:

Heap	Classes	Creation
uncollected	uncollected	<code>new/malloc</code>
copy collected	collected	<code>new</code>
user collected	collected	<code>new(heap)</code>

where we call collected those classes which inherit from `GcObject` and uncollected all others.

With the CMM, object allocation is not tied to the type of an object as in other proposals, so a programmer can design his classes without committing to a particular memory policy. The policy can be decided later, or even be different in different portions of an application. For instance, in the PoSSo solver, one sets the variable `heap` to the heap implementing the stack policy before starting the simplification. Throughout the simplification, all objects (monomial, polynomial, large precision integers, lists and so on) are allocated in this heap and freed in a single step at the end of the simplification. After simplification, one reverts to the normal heap. It is essential that this can be done without changing a single line in the user code.

## 9 Heap Classes

To manage a heap one normally has to maintain the set of roots for the objects in the heap, manage the pages where objects are allocated and implement the memory allocation and recovery primitives. A suitable encapsulation for these functionalities is provided by the `Heap` class.

### 9.1 The Heap Class

A class implementing a heap must supply definitions for the following pure virtual functions: `allocate` and `reclaim`, implementing the memory allocation strategy, `collect` to perform collection, and `scavenge`, the action required to preserve live objects encountered during traversal. Heap classes are derived from the abstract class `Heap`, defined as follows:

```
class Heap
{
public:
    int Index();           // identifies the heap
    Heap();               // initializer

    virtual GcObject* allocate(int ObjSize) = 0;
    virtual void reclaim(GcObject* ObjPtr) = 0;
    virtual void scavenge(GcObject **ptr) = 0;
    virtual void collect() = 0;
};
```

```

// Operations on the Root Set:
void register(GcObject *);      // add an element
void register(GcObject **);
void deregister(GcObject *);   // remove an element
void deregister(GcObject **);
void ScanRoots(Heap *heap);    // scan the roots
bool outside(GcObject *ptr);   // checks if ptr is outside
                                // this heap

void visit(GcObject *ptr) {
    if (! ptr->IsMarked()) {
        ptr->mark();
        ptr->traverse();
    }
}

private:
    int index;
    RootSet *roots;
}

```

`roots` is a pointer to an instance of class `RootSet`, used for registering potential roots. Depending on the particular type of `RootSet` used, the collector can be conservative, type-accurate or both. The simplest `RootSet` considers as possible roots only the objects explicitly registered by the user. The derived class `ConservativeRootSet` scans also the system stack, the process static data area, and registers for possible roots.

The CMM provides three predefined heap classes:

- Bartlett heap: encapsulates the primary collector of the CMM and implements a copying discipline;
- uncollected heap: it provides the standard manual allocation discipline. It is available through the default `new` operator or the functions of the `malloc` library. Objects not inheriting from `GcObject` are allocated in this heap.

## 10 Implementing Heaps

This section illustrates the CMM programmer interface for implementing new heaps. We describe the mechanism through an example, which is a simplified version of the actual heap used in PoSSo.

### 10.1 The HeapStack

A foremost algorithm in the PoSSo algebra system is the one for computing of the Gröbner basis of a set of polynomials. Dependencies between temporaries and persistent data make the use of explicit memory allocation/deallocation nearly

impossible, so use of a garbage collector was essential. The main step of the Buchberger algorithm [6] consists in the simplification of a polynomial which involves many operations creating a lot of intermediate polynomials of which only the last one is relevant and is inserted into the basis. Once this polynomial has been computed, all the temporary structures allocated can be removed.

The peculiar dynamics of the problem offers an opportunity to try out the CMM facilities to implement a specific memory management. We created a heap in which the allocation is stack-like (and thus fast), and the garbage collector called synchronously after each step.

We present a simplified solution in which the size of the stack is fixed, and a copying collector which uses two areas. The real solution we adopted for the problem is more complex and uses a list of areas, and a copying collector.

## 10.2 The HeapStack

First we define the `HeapStack` class as a `Heap` consisting of two areas which implement the `FromSpace` and the `ToSpace` of the collector, and a `RootSet` to register the roots to use for the collection:

```
class HeapStack: public Heap
{
    public:
        void scavenge(GcObject **ptr);
        GcObject* allocate(int words);
        void reclaim(GcObject* ObjPtr) {};
        void collect();
        HeapStack(int size = 100000);

    private:
        pages FromSpace, ToSpace;
        int FromTop, ToTop;
}

HeapStack::HeapStack(int StackSize)
{
    FromSpace = allocate_pages(StackSize, index);
    ToSpace = allocate_pages(StackSize, index);
}

inline GcObject* HeapStack::allocate(int size)
{
    int words = BYTESToWORDS(size);
    int *object = FromSpace + FromTop;
    if (words <= (FromSize - FromTop)) {
        FromTop += words;
        return (GcObject *)object;
    }
}
```

```

    }
    else return (GcObject *)NULL;
}

```

The collector uses the root set to traverse the roots using its traversing strategy. After having moved to `ToSpace` all the objects reachable from the roots, it traverses those objects in order to move all further reachable objects. The specific action required for scavenging objects is as follows:

```

void HeapStack::scavenge(GcObject **ptr)
{
    GcObject **OldPtr = ptr;

    if (OutsideHeap((int *)*ptr)) return;
    GcObject *p = GetBeginning((int *)*ptr) ;
    if (outside(p)) visit(p);
    else if (*ptr->forwarded()) ToBeForwarded(ptr);
    else {
        *ptr = moveTo(ToSpace, *ptr);
        OldPtr->SetForward(*ptr);
    }
}

```

This code relies on support provided by classes `GcObject` and `HeapStack`. As the final step the collector exchanges the roles of `FromSpace` and `ToSpace`.

```

void HeapStack::collect()
{
    pages *TmpSpace;
    GcObject *ObjPtr;
    // Throughout this collection use our scavenge:
    ::scavenge = (void (*)())&scavenge;
    // First traverse the objects registered as roots
    ScanRoots(this);
    // Now traverse the objects already moved into ToSpace
    ObjPtr = ToSpace;
    while (ObjPtr < ToSpaceEnd) {
        ObjPtr->traverse();
        ObjPtr = ObjPtr->next();
    }
    // swap FromSpace and ToSpace
    TmpSpace = FromSpace; FromSpace = ToSpace; ToSpace = TmpSpace;
    FromTop = ToTop; ToTop = 0;
}

```

In our implementation of the Buchberger algorithm we register as roots of the heap two variables containing the base of polynomials and the list of poly-

nomial pairs which are the only objects which need to be preserved after each simplification step when `collect` is invoked explicitly.

## 11 Assessment

Having chosen to base the design of the collector on inheritance and specialisation proved to be convenient to achieve an open design which can be easily extended. But what are the drawbacks of such choice?

### 11.1 Space Overhead

One objection is that all collected objects must inherit from class `GCObject` which declares a `traverse` as a virtual function and therefore space overhead is added to each object. Some overhead is however inevitable to enable garbage collection. Other solutions either add one word of header to identify the type of objects in the heap or allocate objects in a separate region for each type, which also cause some waste of memory. On the other hand we were able to avoid any space overhead except what C++ needs for implementing objects with virtual functions and 2 bits per word in global tables.

### 11.2 Overloading `new`

The CMM exploits the placement syntax of the `new` operator to determine where and how to allocate an object. This of course limits user code from using this feature, even though just for collected objects. It is hard to assess how bad is this limitation: in fact Ellis and Detlefs [10] argue that there is no reason to overload `new` for a collected class, whose allocation is performed by the garbage collector methods. On the other hand, in the CMM, class `GCObject` is just a class defined in a public library whose code is accessible, so there is no difficulty in defining derived classes from it, with suitable specialisation of its `new` operator.

One case in which we found this useful was to define collectable objects of variable size.

Consider the following example:

```
class BigNum {
  BigNum(int l) {
    length = l;
    limbs = new int[l];
  }
  int length;
  int *limbs;
}
```

With this definition of `BigNum`, creating a `BigNum` will require two memory allocations and accessing the array `limbs` will require an extra pointer indirection.

To improve this solution, one may define the class `GcVarObject` as a derived class from `GcObject`, which provides a `new` operator with an additional parameter for the size of the variable member. The following example illustrates its use:

```
class BigNum : public GcVarObject
{
    int length;
    int limbs[1]; // size determined at object creation
};

BigNum *num;
BigSize = 256 * sizeof(int);
num = new(BigSize) BigNum;
```

The object `num` is created in the default heap and has room for 256 integers. The implementation of class `GcVarObject` might be however compiler dependent.

### 11.3 Array of GcObjects

Recently the ANSI standard committee has introduced overloading of the `new[]` operator for allocating arrays of objects. Using this feature, we could define the `new[GcObject]` operator so that an array of `GcObjects` is allocated in the collected heap. Right now, if we define

```
class Person: GcObject {
    char *name;
    int age;
}

Person *Table = new Person[n];
```

the compiler will invoke the system `malloc` to allocate memory from the uncollected heap for `Table` and all of its elements.

For the time being the CMM provides the `GcArray` template class allocating arrays in the collected heap, as in this example:

```
GcArray<Person> &Table = new(n) GcArray<Person> ;
```

### 11.4 Performance

We have received quite satisfactory reports on the performance of CMM by the partners in the PoSSo project who used it in particular for implementing a linear algebra package [12].

To compare the performance of the CMM and the original Bartlett's implementation, we run several classical test cases for the Buchberger algorithm. We report here the results of a couple of these (known in the literature as Katsura5 and Valla), providing details on the timings of memory operations: `alloc`, the

primitive allocator; **scavenge**, the primitive which copies objects to the new space; **gc**, overall time spent in garbage collection; **gc calls**, the number of calls to the collector. Times are in seconds on a SparcStation 10 with 32 Mbytes of memory:

	<b>Bartlett</b>	<b>CMM default</b>	<b>CMM HeapStack</b>
Katsura5	30.7	35.9	22.5
<b>alloc</b>	4.95	8.3	0.83
<b>scavenge</b>	1.9	1.36	2.2
<b>gc</b>	4.51	3.0	3.1
<b>gc calls</b>	178	218	54
Valla	112.0	112.0	71.6
<b>alloc</b>	28.98	32.25	9.3
<b>scavenge</b>	10.64	5.26	3.36
<b>gc</b>	19.74	10.86	8.73
<b>gc calls</b>	506	502	118

There are two factors which contribute to the improvement in performance: less time spent in collection and less time spent in allocation. While the overall improvement is approximately 38% in both examples, the contribution to the improvement is split differently: 90% improvement in allocation and 0% in collection with Katsura5, 75% improvement in allocation and 20% in collection with Valla.

It is also interesting to notice that the CMM default algorithm has similar performance to Bartlett's original, despite the overhead due to its use of member functions.

## 12 Related Work

The Boehm-Weiser collector [5] is a well known collector for C++ which is totally conservative and therefore quite convenient to use. However it is not customisable and is subject to unduly retention of space and memory fragmentation since it cannot compact memory. Our copying collector has some advantage in performance not having to reconstruct a free list after collection and being more accurate in tracing live objects.

Work on adding garbage collection to C++ has been done by D. Samples and D. Edelson. Samples [13] proposes modifying C++, to include a garbage collection environment as part of the language. This may be a good long term approach for garbage collection in C++ but is not suitable for a project like PoSSo which needs portable garbage collection facilities as soon as possible. Our feeling is that this work demonstrates how the flexibility of object oriented

languages can be used to implement a very complex environment, like CMM, without requiring modifications to the language.

Edelson [8] has been experimenting with the coexistence of different garbage collection techniques. The flexibility of the solutions he adopts in his approach allows the coexistence of different garbage collectors, but he does not provide any interface to the user to customise and/or define his own memory management facilities.

Ellis and Detlefs [10] propose some extensions to the C++ language to allow for collectable object. The major change is the addition of the type specifier `gc` to specify which heap to use in allocating the object or a class. They also propose to change the operator `new T` to call the collector allocator when `T` is a `gc` type, and as a consequence of this, the overloading of `new` and `delete` operators for `gc` classes is forbidden. While the `gc` keyword is compatible with our solution of inheriting from the base class `GcObject`, the constraint on `new` needs to be relaxed to allow overloading of `new` when additional arguments are present. Otherwise this constraint will block the possibility of using different heaps for the same kind of objects in different portions of a program. Other suggestions from the Ellis-Detlefs proposals are quite valuable, for instance making the compiler aware of the garbage collection presence and avoid producing code where a pointer to an object (which may be the last one) is overwritten. This can happen for instance in optimizing code for accessing structure members.

## 13 Conclusion

The CMM offers garbage collection facilities without significant compromises. Programmers can use a generic collector, a specific collector or no collector at all, according to the need of each algorithm. The algorithm can be in control when necessary of its memory requirements and does not have to adapt to a fixed memory management policy.

The CMM is implemented as a C++ library, produced with extensive revisions from the original Bartlett's code. It is being heavily used in the implementation of high demanding computer algebra algorithms in the PoSSo project. The CMM provides the required flexibility without degradation in performance as compared to versions of the same algorithms performing manual allocation.

The next challenge would be to incorporate in the C++ compiler the minimal facilities required for CMM support: the addition of the `gc` keyword, proposed by Ellis and Detlefs, could facilitate this.

## 14 Availability

The sources for CMM are available for anonymous ftp from site `ftp.di.unipi.it` in the directory `/pub/project/posso`. Please address comments, suggestions, bug reports to `cmm@di.unipi.it`.

## 15 Acknowledgements

Carlo Traverso and John Abbott participated to the design. J.C. Faugere provided the idea for this work. Joachim Hollman and Fabrice Rouillier helped in testing the first prototype implementation. Discussions with J. Ellis were useful to ensure compatibility of his proposal with our framework. Comments from L. Semenzato helped to improve the presentation.

## References

1. J.D. Ichbiah et al. "Rationale for the design of the ADA programming language", ACM SIGPLAN Notices, 14(6), 1979.
2. G. Attardi and T. Flagella "A customisable memory management framework", Proceedings of USENIX C++ Conference 1994, Cambridge, Massachusetts, April 1994.
3. J.F. Bartlett "Compacting garbage collection with ambiguous roots" Tech. Rep. 88/2, DEC Western Research Laboratory, Palo Alto, California, February 1988.
4. J.F. Bartlett "Mostly-copying collection picks up generations and C++", Tech. Rep. TN-12, DEC Western Research Laboratory, Palo Alto, California, October 1989.
5. H.J. Boehm and M. Weiser "Garbage collection in an uncooperative environment", Software Practice and Experience, 18(9), 1988, 807-820.
6. B. Buchberger, "Gröbner bases: an algorithmic method in polynomial ideal theory", *Recent trends in multidimensional systems theory*, N. K. Bose, ed., D. Reidel Publ. Comp. 1985, 184-232.
7. D. L. Detlefs, "Concurrent garbage collection for C++", CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, 1990.
8. D.R. Edelson "Precompiling C++ for garbage collection", in *Memory Management*, Y. Bekkers and J. Cohen (Eds.), Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 299-314.
9. D.R. Edelson "A mark-and-sweep collector for C++", Proc. of ACM Conference on Principle of Programming Languages, 1992.
10. J.R. Ellis and D.L. Detlefs "Safe, efficient garbage collection for C++", Xerox PARC report CSL-93-4, 1993.
11. G. Nelson, editor "Systems Programming with Modula3", Prentice Hall, 1991.
12. F. Rouillier "Personal communication", 1994.
13. A.D. Samples "GC-cooperative C++", Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 315-329.
14. D. Weinreb, D. Moon and R.M. Stallman "Lisp Machine Manual" Massachusetts Institute of Technology, Cambridge, Massachusetts, 1983.
15. E. P. Wentworth "Pitfalls of conservative garbage collection", Software Practice and Experience, 20(7), 1990, 719-727.
16. P.R. Wilson "Uniprocessor garbage collection techniques", in *Memory Management*, Y. Bekkers and J. Cohen (Eds.), Lecture Notes in Computer Science, n. 637, Springer-Verlag, 1992, 1-42.
17. B. Zorn "The measured cost of conservative garbage collection" Technical Report CU-CS-573-92, Department of Computer Science, University of Colorado at Boulder, 1992.

This article was processed using the L<sup>A</sup>T<sub>E</sub>X macro package with LLNCS style