Comparison of Garbage Collector Prototypes for C++ Applications

Hamid Mcheick, PhD Computer Science
Department of Computer Science and Mathematic
University of Quebec at Chicoutimi, UQAC
555 Boulevard de l'Université Chicoutimi, G7H2B1, Canada
hamid_mcheick@uqac.ca

Aymen Sioud, PhD Student
Applied Science Department
University of Quebec at Chicoutimi, UQAC
555 Boulevard de l'Université Chicoutimi, G7H2B1, Canada
Aymen_sioud@uqac.ca

***Abstract*—In wide-enterprise information system, a Garbage Collector (GC) is a critical memory management process in programming languages such as C++. The programmers need to be aware about memory management and can't be relieved from this task. They have to implement explicitly this task whatever the size of C++ applications. Consequently, this leads to errors and bugs (i.e. dangling pointers, allocation failures, etc.). Therefore, supporting implicit memory management based of aspect-oriented programming can provide a number of benefits such as freeing the programmer of memory management task. We have tackled this problem using aspect-oriented programming to manage implicitly the allocation and release of memory. In this paper, we describe two prototypes of memory management that implement reference counter and mark and sweep techniques. Then we compare them based on the constraints of Detlefs.**

I. INTRODUCTION

Researchers and practitioners have noticed that an object lifecycle management of programs can be handled using either an implicit or explicit strategy [4], [6]. In some programming languages such as Java, Lisp, and Eiffel, the garbage collection (GC) handles implicitly and transparently the object lifecycle and the programmer is relieved from this task. However, in C/C++ and Ada languages, object lifecycle management has been done explicitly and manually [5]. The explicit management of object lifecycle requires a considerable effort from the programmers, consequently leads to errors and bugs (i.e. dangling pointers, allocation failures, etc.). Despite its capacity to manipulate pointers directly, the explicit management implies a heavy task of programmers. For example, Ellis et Detlefs [7] have shown that programmers spend more than 40% of their time to debug errors related to memory management using explicit memory management of C++. However, automatic GC makes memory management transparent and secure.

Several researches such as smart pointer [9] and conservative algorithm [3] have been done to transform explicit memory object lifecycle management to an implicit (automatic) memory lifecycle object management. Most of the object lifecycle management approaches in C++ have major limitations such as fragmentation problem, changing the code generator, etc. Detlefs has specified the following constraints to integrate a useful GC in a C++ application [6]: i) GC requires no compiler support, ii) it requires no information from the programmer regarding the format of GC objects, iii) it has strategies to cope with bugs caused by aggressive optimizing compilers, iv) it allows the use of both automatically and explicit managed storage in the same program, and finally v) it invokes destructors on called heap objects. Therefore, it's necessary to take into account theses constraints without any changes of existing class libraries and C++ implementations.

Figure 1 illustrates new and delete operations to manage explicitly memory. It is clear that the programmer might make some mistakes resulting in bugs and forget to call a delete operation: the programmer does not make delete(p2) at the end of main function.

The principles of these algorithms require instrumentation of code to intercept the calls of memory management operations (go through the heap, find new and delete operators, etc.). This instrumentation can be handled easily by aspect-oriented approach which allows decomposition in terms of aspects. These aspects crosscut classes and modules [Harrison et al., 02] [1, 2] and so far an object lifecycle management can be handled as a technical aspect.

In this paper, we compare our two prototypes which implicitly manage the memory allocation and release of C++ applications.

The next section describes garbage collection algorithms and aspect-oriented approaches. Section 3 describes the first prototype ASPECTGCRC based on the reference counter technique. Section 4 describes the second prototype ASPECTGCMS based on the mark and sweep technique. Comparison between theses prototypes will be given in the section 5.

```
class Point {
    int X; int Y;
    . . .
     Point (int NewX=0, int
     NewY=0){
          X=NewX;Y=NewY;
     }
     void showInfo(){
          cout << X << '\n' << Y;
  }};
int main (){
          Point *p1 = new
Point(10,10);
          p1->showInfo();
          delete (p1);
          Point *p2 = new
          Point(200,200);
          p2->showInfo();
                  }
```

Figure 1. Example of class Point using new and delete operations.

II. Background

A. Garbage Collection Algorithms

Reference counter. The object reference counter consists of maintaining a counter of the number of objects referring to a shared object [8]. This counter is incremented every time a new reference to the shared object is created [5]. Also, this counter is decremented each time a reference to the shared object is destroyed. The advantages of this algorithm are: i) each unreachable object is destroyed by the GC, ii) the algorithm is concurrent, it operates in the same time of the running the program user. The disadvantages are: i) inability to reclaim cyclic structures, ii) this algorithm is too slow because it has to handle the reference counter of each object, and iii) finally the memory fragmentation. We developed ASPECTGCRC prototype based on this technique.

Mark and Sweep. All reachable objects from the roots are marked with a flag in this first step. But the GC doesn't destroy the objects in this step. In the second step, the GC goes throw the heap and destroys all unmarked objects and then it will unmark all marked objects [5]. By default, all objects are unmarked when they are created. It's evident that this algorithm can transcend the problem of the inability to reclaim cyclic structures. Also this algorithm offers a better performance than reference counting in time consumption. Nevertheless, the cost of the mark and sweep algorithm is higher because its asymptotic complexity is proportional to the size of the entire heap. Another disadvantage of this technique is the fragmentation caused by reclaiming dispersing objects in the heap. We developed ASPECTGCMS prototype based on this technique using aspect-oriented programming to instrument C++ code.

B. Aspect-Oriented Programming

Aspect-oriented programming (AOP) recognizes that the programming languages that we use do not support all of the abstraction boundaries in our domain models and design processes. Underlying AOP is the observation that what starts out as fairly distinct concerns at the requirements level, or at the design requirements level (non-functional requirements) end us tangled in the final program code because of the lack of support, both at the design process level, and at the programming language level, for keeping these concerns separate. With AOP, these concerns may be packaged as aspects, which can be woven into "any" application that has those concerns.

In order to clearly understand the concepts of AspectC++™, we will explain theses concepts through the example in Figure 2, where we have construction and destruction of objects. In this example, we have an aspect, which calculates the number of an object created instances.

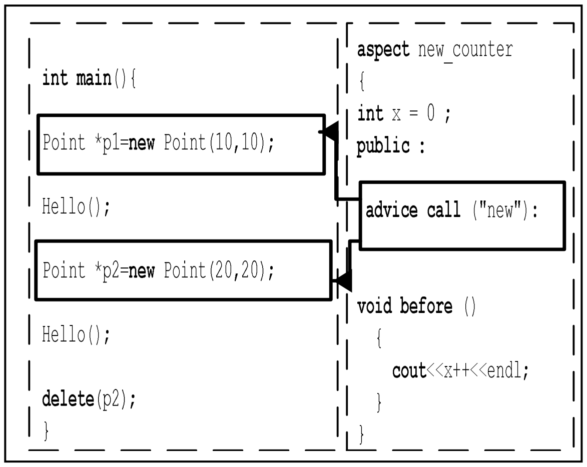
Here we give an example of before/advice, which counts the number of occurrences of the new function's calls. The following aspect new_counter counts theses occurrences:

```
aspect new_counter {
 int x = 0 ;
 public : // list of pointcuts when
          //we find new operator
  advice call ("new"): void before ()
   {
    Cout << x++ <<endl;
   }
}
```

So, as shown in the Figure 3, we can see how the aspect new_counter is transplanted in the example of the Figure 3. In this aspect, joint point call("new") is made in every call of the function new. The advice before() contains the portion of code which will be executed in every call of the function new. And so we can count the number of occurrence of the calls of this function.

```
class Point {
  int X; int Y;   ...
  Point (int NewX=0, int newY=0)
    {X=NewX;Y=NewY;}
  void Affiche()
    {cout << X << endl << Y;}};
void Hello(){
     cout << "Hello" << endl; }
int main (){
 Point *p1 = new Point(10,10);
 Hello();
 delete (p1);
 Point*p2 = new Point(200, 200);
    Hello();
    delete (p2);
}
```

Figure 2. Example of class Point before using weaving.



```
int main(){
Point *p1=new Point(10,10);
Hello();
Point *p2=new Point(20,20);
Hello();
delete(p2);
}
```

```
aspect new_counter
{
int x = 0 ;
public :
advice call ("new"):
void before ()
  {
   cout<<x++<<endl;
  }
}
```

Figure 3. Example of class Point before after weaving.

UQAC and NSERC Canada

III. ASPECTGCRC: A PROTOTYPE OF GARBAGE COLLECTION USING REFERNCE COUNTER TECHNIQUE

To manage object lifecycle in C++ legacy applications, we propose a prototype (AspectGCRC) based on AspectC++™ and the reference counter technique. AspectC++™ uses a weaver to integrate C++ applications and object lifecycle management into C++ standard applications. In this section, a concept and implementation of reference counter is presented.

A. *Concept of AspectGCRC*

AspectGCRC idea consists of developing an automatic implicit technique to manage object lifecycle in C++ applications. This management is important to support performance criteria which are needed in different domains. Using AspectC++™ tool as a weaver, the output is an integrated C++ applications where reference counter are implicitly maintained. In this section, we explain the implementation of AspectGCRC as well as the object table of AspectGCRC.

The implementation of AspectGCRC prototype is done in two phases. In the first phase, the prototype scans the application and retrieves all application class definitions including user created classes. In the second phase, retrieved classes are scanned for object creation. For each class the prototype maintains in an object table (see table 1) the name of the class and a list of references to the object created of this class. In addition a reference counter of the number of objects created is maintained so that appropriate handling can be taken when new objects are created or older objects are no longer useful and are ready for garbage collection. The reference counter is incremented when a static instantiation, a reference assignment or a constructor call occurs. Furthermore, when a call to a destructor, delete or free is encountered the reference counter is decremented. Finally, the prototype scans the object table for reference counters having a zero value and deletes the object associated with this reference counters. Also, we decrement the reference counters for objects, which maintain a reference to deleted object. Finally, we delete the objects that have zero as reference counter (memory leaks) and therefore we skip the delete and destructor instructions of theses objects (because if an object is deleted then delete instruction is unused). Traditionally, garbage collector traverses a graph of objects using registers and classes as roots. In our case, we traverse the code using classes as roots. Each created object is an instance of a given class in the table.

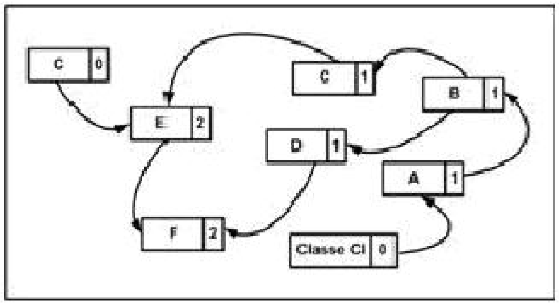


Figure 4. Example of reference counter.

A.1 **Object Table**

The object table is a data structure, which maintains valuable information useful to the implicit garbage collector implementation. Table 1 shows the attributes of this data structure:

1) *Nom_obj* designates the name of the object or class. This attribute is returned by an advice, which contains the member function of JointPoint class (thisJointPoint->toString).
2) *Cls* is a boolean flag to determine if this is a class or not.
3) *Cpt_obj* is reference count associated with this object.
4) *List_ref_A* designates the list of references, which point to this object.
5) *List_ref_de* designates the list of object to which this object refers to. The last two previously mentioned attributes are updated each time a new object owning a reference to the object associated with the reference counter.
6) *Pro_const* is a list containing the prototypes of all existing constructors associated with this object. We also use the member function of the class JointPoint: Args (type pattern, …)to return all the constructors.
7) *Pro_dest* is the prototype of the destructor associated with the returned object by the function Args (ptr_object)which is a member function of the JointPoint class.
8) *Sur_op* is a boolean variable to indicate if the operator " = " is overloaded or not.

TABLE I. OBJECT TABLE STRUCTURE

Nom_obj	Cls	Cpt_obt	List_ref_a	List_ref_de	Pro_const	Po_dest	Sur_op

The list of referring objects List_ref_a maintains the list of objects referring to the object or class under study. Each time an object adds a reference to the object under study, the reference will be added to this list and the reference counter will be incremented. To improve the efficiency of our prototype, we apply a deeps first search to solve cyclic problem inside a graph of objects. The different objects represent the nodes and List_ref_a represents a list adjacency of different nodes.

TABLE II. OBJECT TABLE AFTER PASSING ACROSS CLASSES

Nom_obj	Cls	Cpt_obt	List_ref_a	List_ref_de	Pro_const	Po_dest	Sur_op
Point	true	-	-	-	Point(int,int)	-	true
Pixel	true	-	-	-	Pixel(int,int,int)	-	false

A.2 **Development Process of AspectGCRC Using Sample Example**

In figure 4, the deletion of the object, A which is an instance of the class C1, decreases the reference counter of B. This last counter decrements the respective reference counters C and D since the reference counter of B became 0. In a second step, the reference counters E and F will be decremented consequently.
To illustrate the way the reference counter technique work, we apply our technique on the example of figure 5. First, we pass across the class Point (L1 to L9). The boolean attributes Cls and Sur_op are true. In addition we have a constructor prototype Pro_const : Point(int,int). After passing across the class Pixel (L10 to L18), we will have in the object table Cls true and, Sur_op false and a constructor Pro_const : Point(int,int,int). These two classes do not have any constructors. Finally, we obtain the object table illustrated in table 3.
Lines L20 and L21 designate a static instantiation of two objects of type Point. The reference counters Cpt_obj associated with P1 and P2 are initialized and the list List_ref_a of the class Point will be updated by adding the objects P1 and P2. The result object table is illustrated in table 3.

TABLE III. OBJECT TABLE AFTER INSTANTIATION OF P1 AND P2

Nom_obj	Cls	Cpt_obt	List_ref_a	List_ref_de	Pro_const	Po_dest	Sur_op
Point	true	-	P1,P2	-	Point(int,int)	-	true
Pixel	true	-	-	-	Pixel(int,int,int)	-	false
P1	false	0	-	-	-	-	-
P2	false	0	-	-	-	-	-

Line L22 represents an instantiation and assignment of an object of type Point. The reference counter of P3 is initialized. The reference counters of P1 and P2 are incremented. Similarly, P3 is added to the list List_ref_a of the class Point. Lines L24 to L27 represent the declaration of pointers to objects of type Pixel. At line L30, the object P1 is deleted as well as the pointers to the objects ptrPixel1 and ptrPixel2.
Table 4 illustrates the object table at program end after the deletion of these 3 objects.
At the end of the program, the reference counters of the objects ptrPixel3 and ptrPixel4 have a value of 0; therefore they have to be deleted.
Similarly, the reference counter associated with P3 is also 0, therefore P3 is to be deleted. Before deleting P3, we pass across the list List_ref_de to determine the list of objects referring to P3. P2 is included in the list List_ref_de of P3, the reference counter associated with P2 is therefore decremented, which will become 0 and result in the deletion of the P2 object. In the case of a deletion of the P3 object with the delete operator and another deletion of the P2 object, the invocation of the delete operator on the P2 object will be cancelled via a

call to the void proceed() function simply because its reference counter becomes 0 and therefore the object need be deleted. This last function calls the around() advice.

```
class Point {
    protected: int X ; intY;
    public :
    int GetX(void) {return X;}
    int GetY(void) {return Y;}
    Point (int NewX=0, int NewY=0) {X = NewX; Y = NewY;}
    Point operator+ (Point &P1, Point &P2)
    {Point res(P1.GetX()+P2.GetX(),P1.GetY()+P2.GetY(),P1);
     return res;}
    Point operator = (Point &P);
    {Point res; res.X = this.X; res.Y=this.Y; return res;}
};
class Pixel : public Point {
    protected:    int couleur;
    public :
    Pixel (int nx,int ny,int coul=0):Point(nx,ny)
    Pixel (int nx,int ny,int coul):Point(nx,ny) {couleur=coul;}

    void allume(void); {g_pixel(X,Y,couleur);} //une fonction qui
    allume un pixel à l'écran
    void allume(int couleur); {g_pixel(X,Y,couleur=coul);}
    void eteind(void); {allume(0);}
};
void main(void){
  Point P1(25,25);
  Point P2(20,20);
  Point P3=P1+P2;
  int coordX = P.GetX();
  Pixel *ptrPixel1 = new Pixel(100,100,1);
  Pixel *ptrPixel2 = new Pixel(200,200,2);
  Pixel *ptrPixel3 = new Pixel(300,300,3);
  Pixel *ptrPixel4 = new Pixel(400,400);
  ptrPixel1->allume();
  ptrPixel1->eteind();
  delete P1;
  delete P2;
  delete ptr1Pixel;
  delete ptr2Pixel;
}
```

Figure 5. C++ Example for refernce counter.

IV. AspectGCMS: A PROTOTYPE OF GARBAGE COLLECTION USING MARK AND SWEEP TECHNIQUE

We describe in this section the AspectGCMS prototype.

A. *Concept of AspectGCMS*

The idea of AspectGCMS consists of developing an automatic implicit technique to manage object lifecycle in C++ applications based in the Mark and Sweep technique. This management is important to support performance criteria which are needed in different domains. Using AspectC++™, as weaver, the output is a standard C++ application where garbage collection approach is implicitly maintained.

Traditionally, garbage collector traverses a graph of objects using registers and classes as roots. We traverse the code using classes as roots. We use Depth-First-Search (DFS) to go through the object graph in order to mark reached objects and to update the number of the garbage collector visitors. Then we use the slide compact method and we copy the object in a free space allocation. Finally, to update the object pointers, we use the Threaded method. This method needs a list of objects to keep their orders.
In next section, we explain the implementation of AspectGCMS.

B. *Implementation of AspectGCMS*

The implementation of the AspectGCMS prototype is done in two phases. In the first phase, it scans the application and marks all the reachable objects starting from the root. In the second phase, the unmarked objects are added to pool of free objects. Here is an example of mark step using a pointer on the root. This example is written inside an aspect which is integrated with any C++ application.

```
aspect GCMarkSweep {
     void Pointers::markPointer (void*
     pointer)
     {
       Pointers* current = head;
       ...
       current->content->marked = true;
     }
     ...
     void GCMarkSweep::sweep() {...}
}
```

We use a data structure to handle the object created by C++ applications. This structure consists of an aspect (GCMarkSweep) and two classes (Pointers and Pointer) to mark the reached objects. Pointer is a wrapper, which contains a flag to mark theses objects. The class Pointers contains the list of allocated objects.

```
class Pointer {
     bool marked;
     int count;
     ...
}

class Pointers { ...
     static Pointers* head;
     Pointers* next;
     Pointer* content;
     ...
}
```

This trace identifies active objects and deletes unused ones. This implementation is based on AspectC++™, a weaver of C++ application with aspect. This aspect is developed independent and applied on C++ legacy applications. In fact, mark and sweep is implemented in AspectGCMS which will be extended to implement Mark and Compact later in future work.
The following aspect shows a specification to handle new and delete operations into two advices of AspectC++™ language. Theses advices are used during runtime to get a point of execution in the code. This point of code execution is called a jointcut, which links an aspect with C++ classes. This aspect matches theses operations into any C++ application without modification its code, even if this application contains some explicit memory management code. This aspect creates an independent structure to handles memory objects. Therefore, it will not be inserted into C++ application but marks and sweeps the unreferenced objects. Theses steps use marked and count variables specified in section 2.2.

```
aspect GCMarkSweep {

 advice (execution("% ...::operator
 new(...)")): after (){
   GCMarkSweep::aspectof()->
   addPointer(…);

  advice (execution("% ...::operator
  delete(void*)") || execution("%
  operator delete(void*)")): before(){
    GCMarkSweep::aspectof()->
    removePointer(…); }
}
```

V. COMPARISON

To be useful, C++ garbage collection should satisfy the following five constraints [6]:

- Minimal changes: Too many or too severe changes to the language, its implementations, or programming styles will impede acceptance of garbage collection by the C++ community.
- Coexistence: Program components using garbage collection must coexist with components not using it.

- Safety: The rules for correct use of garbage collection should be explicitly defined, and the language and its implementation should provide optional automatic enforcement.
- Portability: A program using garbage collection should run correctly on all implementations of C++.
- Efficiency: The more efficient garbage collection is, the more likely it will be accepted.

Our two prototypes, AspectGCRC and AspectGCMS meet the first three constraints. Indeed, programmers will not have to change their code; the two prototypes use the wrapping technique for the reference counter or the mark flag. The fact that we use aspect programming will not affect the handling with other components not using garbage collection and will ensure the safety.

The portability depends on the platform using the weaver and she conflicts with efficiency and minimal changes. For example, C++ allows a pointer to be cast to a sufficiently large integer and back again, yielding the same pointer. If we want to allow implementations the freedom of using potentially more efficient algorithms (like those of AspectGCRC and AspectGCMS), then the safe-use rules must encompass those algorithms by restricting the use of certain C++ features.

To be successful, garbage collection needn't be quite as efficient as programmer-written deallocation, since many programmers would gladly sacrifice a little extra run time or memory to eliminate storage bugs quickly and reliably. Zorn' measurements indicate that garbage collectors can often be as fast as programmer-written deallocation, sometimes even faster [Zorn, 92]. Just as many programmers think they can eliminate all storage bugs, they also think they can fine-tune the performance of their memory allocators. But in fact, any project has a finite amount of programming effort, and many, if not most, programs are shipped with imperfectly tuned memory management. This makes garbage collection more competitive in practice. Efficiency conflicts with minimizing language changes and enabling coexistence. Most previous approaches to garbage collection have relied on non-trivial language support to achieve acceptable performance.

Our first tests showed that AspectGCMS consumes less time then AspectGCRC due to the subroutine which identifies the graph cycle. The memory consumption is almost identical for the two prototypes but they suffer from memory fragmentation and we need more tests to verify the impact of fragmentation. Also we need more investigation about the attempt of AspectGCMS in the sweep phase.

VI. Conclusion

Our work describes two prototypes of garbage collector for C++ applications. In C++ applications, the programmers need to be aware about the operators of allocation and freeing the memory and therefore it leads to errors and bugs.

Theses prototypes implement respectively reference counter and mark and sweep techniques using aspect-oriented programming. Each prototype has advantages and limits based on Detlefs' constraints (see comparison section). We use aspect-oriented programming (AOP) to ensure the safety and implement implicitly object lifecycle management. We believe that we can add automatically a garbage collector as a component to C++ application using AOP.

In the future work, we need to develop a tool to manage this lifecycle of C++ application and validate it.

Acknowledgment

This work was sponsored by the University of Quebec at Chicoutimi (Quebec), and by NSERC (Canada).

References

[1] G. Kiczales, John Lamping, A. Mendekar, C. Maeda, C.V. Lopes, J.M. Loingtier, & J. Irvin, "Aspect-Oriented Programming," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP07)*, Finland, pp. 220-242, 1997.

[2] A. Rashid, "Aspect-oriented and component-based software engineering," IEEE Proceedings – Software, vol. 148, pp. 87-88, 2001.

[3] H. Boehm, "Bounding Space Usage of Conservative Garbage Collectors", *Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 93-100 Jan. 2002.

[4] A. Alexandrescu, Modern C++ Design, Addison Wesley, 2001.

[5] R. E. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[6] D. L. Detlefs, Garbage collection and runtime typing as a C++ library. In *USENIX C++ Conference*, Portland, Oregon, USENIX Association. August 1992.

[7] J. R. Ellis and D. L. Detlefs, Safe, efficient garbage collection for C++. *Technical report*, Xerox PARC, Palo Alto, CA, 1993.

[8] E. Frank, DCOM: Microsoft Distributed Component Object Model, Redmond III November, 1997.

[9] D. R. Edelson and I. Pohl, A copying collector for C++. In *Usenix C++ Conference Proceedings*, pages 85-102. USENIX Association, 1991..