A Novel Design of A Generational Garbage Collector

ADEEL ABBAS, AFFAN AHMED, WAHEED UZ ZAMAN BAJWA
Electrical Engineering Department,
College of Electrical and Mechanical Engineering,
National University of Sciences and Technology, Rawalpindi, Pakistan

Abstract: In this paper, we describe an efficient and flexible generation scavenging garbage collector toolkit. Our methodology exceeds in systems where the cost associated with any single operation is bounded by small time constant. The collector maintains several "generations" of objects. Newly created objects are all put in the "youngest generation" and when the space allocated for that generation is full, the collector will use the root set to reclaim dead objects from the youngest generation only, leaving the older generations untouched.

Objects that survive after several collections of the youngest generation are "promoted" to the older generation. In this way the collector assures that recently created regions will contain high percentages of garbage and will be garbage collected frequently. The framework has been implemented in a library and tested against formidable real world applications. Hence the proposed system eliminates the possibility of storage-management bugs and making the design of complex, object-oriented systems much easier. This can be accomplished with almost no change to the language itself and only small changes to existing implementations, while retaining compatibility with existing class libraries.

Key Words: Garbage collection, generation scavenging, heap management

1. Introduction

Dynamic memory management is often referred to as making memory requests from the operating system during different courses of program execution. All dynamic memory requests are satisfied through an area of memory called *heap*. In C++ this is done through the **new** operator. The operator is implemented by a call to *malloc()*, which grants a pointer to the object after allocating its memory on the heap. However this flexibility of acquiring memory dynamically comes at a price: i.e. it becomes the responsibility of the programmer to return dynamically allocated memory to the free pool, by using the **delete** operator. The delete operator in turn calls *free()*, which reclaims memory allocated on the heap. When delete has been called, the object is destroyed and its destructor has been called. Further access to the pointer data can cause unpredictable results.

The term "Garbage Collection" is an automated process of finding previously allocated memory that is no longer reachable by the program and then regaining that memory for future use. The garbage collector does this by several ways, one of which is traversing all pointers on the heap and finding weak pointers (pointer that allows the object memory to be recovered). In simple terms, use of Garbage Collectors leverages the programmer of worrying about calling delete every time new is called. Automated Garbage Collectors can reduce development cycles for large-scale software by approximately 30% and additionally reduce the memory leaks, resulting in a more stable system.

Some systems also use reference counting for implementing garbage collection, however they have unnerving disadvantages of their own:

1. The inability to reclaim circular structures i.e. circular structures can have non-zero reference counts, even when garbage.
2. Often results in memory fragmentation.
3. It's expensive since every allocation / freeing requires addition/subtraction.

Due to the above-mentioned problems, it is not a viable option to use reference counting as a primary answer to memory management problems especially when program code begins to increase. Nevertheless, there have been very few implementations of garbage collectors available in the public domain. This paper presents a unique methodology of heap allocation that is based upon copying garbage collection. Our work differs from previously reported work [1], [2] and [3] since it addresses garbage collection targeted to C++ systems.

2. Garbage Collection Terms

Garbage collection algorithms have been the subject of intense study, because they play such an important role in the performance of certain systems. Following are some of the terms that are often referred in elaborating garbage collection algorithms:

2.1. Root Set

The data that is immediately available to the program, without following any pointers. Typically this would include local variables from the activation stack, values in machine registers and global, static or module variables.

2.2. Reachable Data

Data that is accessible by following pointers (references) from the root set. Reachability is a conservative approximation of liveness and is used by most garbage collectors.

2.3. Live Data

Data that is reachable and that the program will actually make use of in the future. Garbage Collectors typically cannot tell the difference between live and reachable data, but compilers can.

2.4. Forwarding Pointer

In a collector that moves objects, a forwarding pointer is a reference installed by garbage collector from an old location to a new one.

2.5. Weak Reference/Pointer

A pointer to an object, which does not prevent an object from being reclaimed. If the only pointers to an object are from weak references, the object may disappear, in which case the reference is replaced by some unique value, typically by the language's equivalent of a NULL pointer.

3. The Garbage Collection Methodology

This section describes the algorithm of garbage collection. Our heap is made up of several generations. As objects survive repeated scavenges, they are promoted to older generations. Higher generations are scavenged more frequently.

Once memory allocation request is made, the garbage collector returns a pointer to the object (created in its own heap space). The collector also stores the address of the pointer (created on the processor stack) for future modifications. The addresses of all pointers, which are created via our collector, are maintained in a *vector* of *void***. The size of the object is also recorded and used during generational copying. Moreover the memory for the recently created object is allotted from the youngest generation.

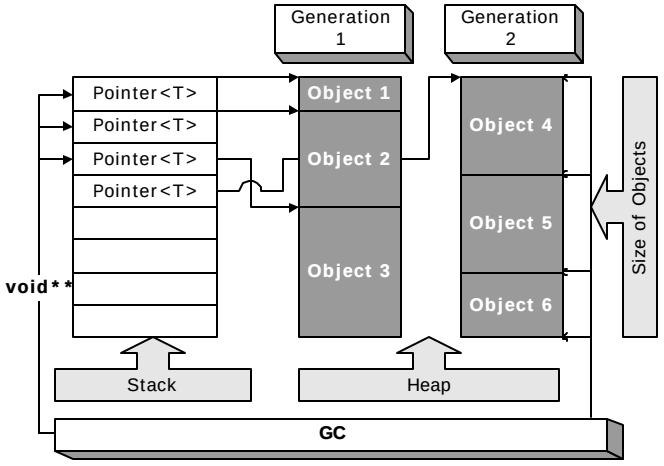

Figure 3.1: The Garbage Collector Design Approach

In order to trace the scope of the pointer, we wrap the pointer returned by the collector in a templatized smart pointer class, *Pointer<Class T>*. Once the smart pointer runs out of scope, or a pointer assignment is made, the garbage collection is run, to verify integrity of all pointers. The garbage collection algorithm involves following steps:

1. The GC iterates all the generations in the heap.
2. For each object in a generation, the collector allocates space in the evacuation region and copies the contents of the old object into the new space.

The collector copies the object in the newer space only if it lies in the reachable data.

3. The collector then re-assigns all pointers on the stack and heap and the processor registers, which contain older address of the object evacuated recently.
4. If no object is evacuated in a generation, then it implies that all the data in the generation is garbage and its memory must be reclaimed.
5. Having a pointer to an object is an indication that the pointer is needed by some other object. Therefore the garbage collector is only allowed to recover an object if no pointers exist to it.

4. C++ implementation

Some of the main classes that implement the collector are:

4.1. GC

GC is the class for garbage collector. It has several static methods that can be used to have direct control over the whole process. The interface of the GC is shown below:

```
class GC
{
private:
// Array of pointers to pointers (made on stack)
static std::vector< void** > _PointersOnStack;
// Holds the size of objects made on the stack
static std::vector< unsigned int > _SizeOfObjects;
// Holds all the generations
static std::vector< Generation* > _Generations;
// Holds total bytes allocated on the heap
static int BytesAllocated;

public:
// Invokes the GC for all generations
static void Collect();
// Invokes the GC up to the generation specified
static void Collect( int Generation );
// Allocates memory from the garbage collector
static void* operator new( size_t, void** pPtr );
// Gets maximum number of generations
static int GetMaxGeneration();
// Gets the total memory allocated on the heap
static int GetTotalBytesAllocated();
// Returns the total number of generations
static int GetGenerationCount();
// Sets the total bytes allocated by the GC
static void SetTotalBytesAllocated( int Value );
};
```

4.2. Pointer<Class T>

In our system, all weak pointers must be objects of type *Pointer<T>*. This class implements the functionality of proxy pointers. It overloads several operators including dereferencing operator, indirection operator, assignment operator and automatic conversion operators. Garbage collection process is invoked whenever a pointer assignment is made. In case the pointer runs out of scope, then collection is invoked and destructor on that object is explicitly called.

```
template <class T> class Pointer
{
// Invoked on assignment and destruction
void Destroy();
public:
T* p; // Wrapped pointer
Pointer( T* p_ = NULL ); // Constructor
~Pointer();// Destructor
// Assignment operator 1
Pointer& operator = (Pointer<T> & p_);
// Assignment operator 2
Pointer& operator = (T* p_);
// Automatic type conversion to T*
operator T*() { return p; }
// Dereferencing operator
T& operator*() { return *p; }
// Pointer indirection operator
T* operator->() { return p; }
// For automatic type conversion during new call
operator void**();
};
```

4.3. Generation

The GC manages its generations on the heap by a class *Generation*. Each generation has a table of contiguous memory locations. Therefore by having newly created objects close together, the program has fewer page faults and the objects will also reside in the processor cache. The generation with is the highest number contains the objects most recently created. Each generation has certain capacity and when the objects on heap overrun that capacity, a new generation is automatically created.

With the framework of generational agglomeration, our system has several advantages, which are:

1. Advancement policy i.e. the collector knows when to consider an object old.
2. Heap organization i.e. the collector can be configured for the number and size of generations the heap should be divided into.
3. Cross-Generational references.

```
class Generation
{
private:
// The generation number
int _GenerationNumber;
// Pointers to the objects in the generation
std::vector< void* > _Pointers;
// Points to the top of memory in the generation
void* _pTopOfMemory;
// Returns maximum size for the generation
static int MaxSize;
// Table of memory inside generation
BYTE MemoryTable[MAXSIZE];
public:
// Gets the remaining memory of the Generation
int GetRemainingMemory() const;
// Returns maximum memory for one generation
int GetTotalMemory() const;
// Grants memory for an object, returns its void*
void* Allocate( size_t Size );
// Gets the generation number
int GetGenerationNumber() const;
// delete operator
void operator delete( void* v );
}
```

5. Experiments And Results

Objects allocated with the built-in "operator new" are uncollectable. Only objects allocated with overloaded new operator that takes address of pointer as the second argument are collectable.

```
void* operator new(const size_t sz, void** pVoid )
{
        return GC::operator new( sz , pVoid );
}
```

The following code demonstrates differences in object creation and usage by the above-mentioned collector:

```
// Traditional approach - memory leaks
int* pInt = new int;

// Our approach - no memory leaks
Pointer<int> pInt = new(pInt) int;
```

The toolkit has been run on various platforms including Unix/Solaris and WinNT/2000. It has been successfully tested with several compilers (Microsoft Visual C++, Borland C++ an GNU). Several algorithms, including DSP algorithms (FIR, IIR Filter etc) and common data structure (link lists, stacks, queues, trees etc) algorithms have been developed using the garbage collector. Using the proposed collector, the development time reduced drastically. We also concluded that the cost of garbage collection was 15-20% of the overall execution time. In order to prove the quality of the collector, we also overloaded global **new** and **delete** operators. Using a simple count of a variable i.e. incrementing it in **new** and decrementing it in **delete**, we were able to show that our collector produced absolutely no memory leaks. Following code demonstrates some of the code that we wrote for testing our collector.

```
#define TEST_COUNT    10000

for( int i = 0; i < TEST_COUNT; i++ )
{
int* pInt = new int;
*pInt = 344;
} // memory leaks

for( int i = 0; i < TEST_COUNT; i++ )
{
Pointer<int> pInt = new(pInt) int;
*pInt = 233;
} // Garbage collection is invoked

// Garbage Collection Test Functions

// The counter is incremented
void* operator new( unsigned int cb )
{
g_Count++;
return malloc( cb );
}

// The counter is decremented
void operator delete( void* v )
{
g_Count--;
free( v );
}
```

6. Conclusions And Future Work

We have introduced a garbage collector framework and presented the interface for such a toolkit. The toolkit approach is itself novel and includes a number of additional innovations in flexibility, performance and interaction between the compiler and the collector. Our future endeavors would focus on testing it against further real world applications. Additionally we would also work on redesigning our collector, in order to make it work in parallel with actual application execution.

References:

[1] H. Lieberman and C. Hewitt, “A real-time garbage collector based on the lifetime of objects,” in *Communications of the ACM 26,* pp 419-429, June 1983

[2] P. Sobalvarro, “A lifetime based garbage collector for LISP systems on general-purpose computers,” MIT, Cambridge, 1988

[3] B. Zorn, “Comparing mark-and-sweep and stop-and-copy garbage collection,” in *Proceedings of the ACM symposium on LISP and functional programming*, France, pp 87-98, October 1989.

[4] L. Deutsch and D. Bobrow, “An efficient, incremental, automatic garbage collector,” in *Proc. Commun. ACM 19*, pp 522-526, September 1976

[5] H. Gao and K. Nilsen, “Reliable General purpose Dynamic Memory Management for Real Time Systems,” in *Proc. IEEE Real-Time Systems Symposium.*