C++ AutoGarbage

An automatic mark and sweep garbage collector for the C++ Programming Language

Michael Lynch

June 18, 2020

Abstract. C++ has become one of the most popular languages in the world, partially through its capability for very deep control over memory management. This project takes advantage of C++ power over direct memory, to implement an automatic garbage collection system similar to those found in more high level languages such as Java or C#.

Contents

1 Treadmill Garbage Collection

AutoGarbage uses the treadmill garbage collection technique[2] to free up memory. This is implemented on a single threaded basis with all marking and sweeping taking place when the user invokes functions from the library.

Currently it is not possible to use AutoGarbage on multi-threaded systems, however treadmill garbage collection can very easily be adapted to work on a seperate thread[B] giving the capability for multi-threaded support in the future.

2 Usage

The use of several classes and macros are required in order for the user to take advantage of the AutoGarbage memory management system. To start AutoGarbage (AGC), the heap needs to be initiated using:

```
1  gc::init_gc(4096, 25);
```

- The first parameter provides the size that the heap should be initiated to in bytes.

- The second parameter indicates how many times the allocator should attempt to allocate an object before giving up and performing a garbage collection.

AGC will heap allocate any object which inherits from the `gc::object` class. AGC expects `gc::object` classes to be formatted with garbage collectable fields (`gc::field`) first, followed by the END_GC_FIELD macro, before standard C++ members in the class' memory map.

```
1  class A : gc::object
2  {
3    // These are garbage collectable objects.
4    gc::field<B> _b;
5    gc::field<C> _c;
6
7    //This notifies that there are no more garbage collectable fields in the object.
8    END_GC_FIELDS;
9
10   //These are not pointer objects so get garbage collected along with the root object.
11   int a;
12   int b;
13 }
```

Listing 1: Example Object

It is important to note that destructors for garbage collected objects are never called (objects just get overwritten once they are collected) so it is inherently unsafe to use any kind of standard C++ pointer (raw or smart) as a member on a `gc::object` unless you explictly manage creation and deletion of the objects yourself.

2.1 Arrays

AGC also supports garbage collectable reference arrays similar to ref arrays found in Java. AGC supports suntime sized arrays through the `gc::object` template class `gc::array<class T>`. As an inheritor of `gc::object`, arrays can be placed into `gc::field` and `gc::static_ptr` just like any other AGC object.
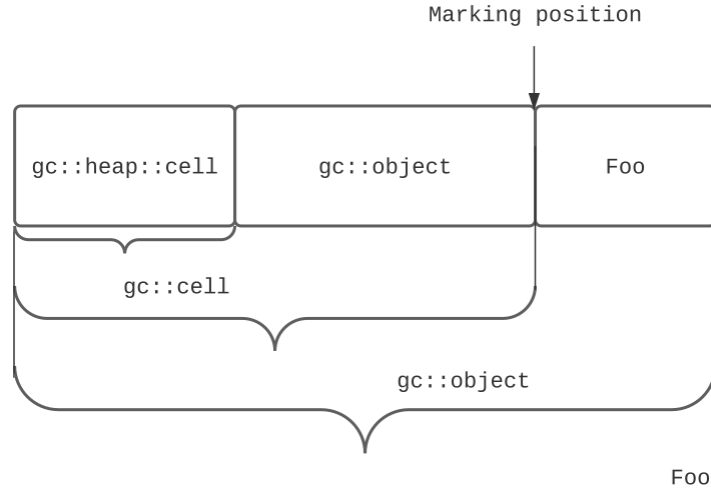
Figure 1: C++ Casting

- `gc::field` should be used to host AGC type members on classes.

- `gc::static_ptr` should be used in the scope of a program accessing AGC type members to ensure that it is not going to be garbage collected before the program has finished using it.

3   Technical Details

3.1   Marking

AGC relies on an obligation with how the users layout their object classes as a work around for C++' lack of runtime reflection. Garbage collectable fields are placed at the top of the class so that the marker knows where to start searching for fields. C++ objects are formatted with the highest class in the inheritance tree placed first. This is done to make it easy to cast one object to another without a dedicated runtime casting checker as shown in Figure 1.

Because of this, and the fact that a class will inherit from the `gc::object` class, the marker is positioned to start at `(((char*)this) + sizeof(gc::object))`. Because fields start at the beginning of the user specified section of the object, this position can be assured to start at the initial field (if there are any). Due to the lack of reflection, the marker needs to work out when to stop marking by analysing the data at the current marking position. This is achieved based on knowledge of how a field should be formatted.

Fields have to hold a reference to an object in the heap of a `nullptr` in order to be valid. Both formats are easy to test:

- If the marker sees a `mullptr` at its current position it will accept this as a valid field and move onto the next one.

- If the marker sees a pointer address which is within the range of the heap it will mark the referenced object and then move on to the next field.
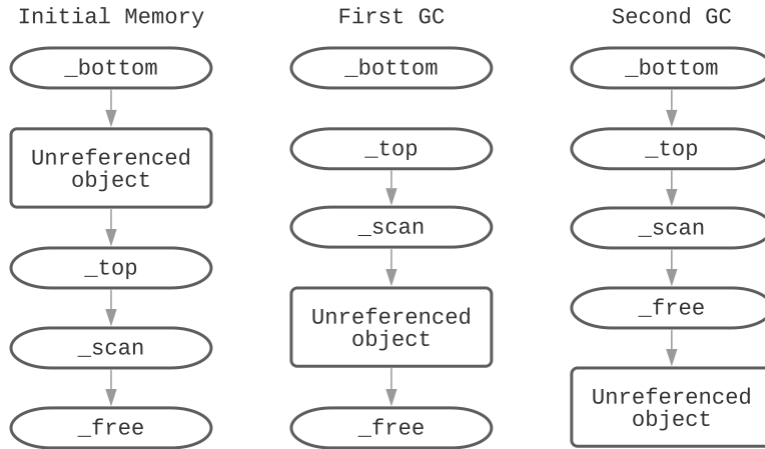
4

Figure 2: Collecting an Unreferenced Object

Fields are ended with the END_GC_FIELDS macro. This macro actually adds a new member to the class of type `void*` pointing to one position less than the allocated heap. The marker will look specifically for this pointer value when looping through the fields to mark. If it sees this value, it will know that it has marked the last field and can stop.

3.1.1 Optimisation Attempts for Stopping Marking

3.2 Allocating

AGC objects have an overidden `new` operator which calls the `gc::heap::heap_struct::malloc` function to provide a suitable memory position in the heap.

3.2.1 `gc::heap::heap_struct::malloc`

The `malloc` function uses the `attempt_allocate` member function to find the memory position. If this function returns a `nullptr`, this means that the function was unable to find a suitable memory position, and the system should attempt a garbage collection before continuing.

`malloc` will attempt to allocate twice after two gc cycles, before giving up and throwing a `bad_alloc` exception. The reason for the double collection is to ensure that recently unreferenced objects have an opportunity to become freed when attempting to allocate as seen in figure 2.

If the system is unable to allocate an object after two gc rounds, it is highly likely that it is not possible to allocate the given object (This does however depend on the number of alloed allocation loops, which will be discussed with regards to the `attempt_allocate` function).

Once `malloc` has been provided a suitable memory position it will re-adjust teh position to take account of V-Table. In C++, objects that make use of virtual functions start their memory allocation with a pointer to a V-Table to perform the function lookup. AGC has two main kinds of structures that appear on the treadmill lists. Initially the treadmill is set up with a single `gc::cell` in the free list which is the same size as the given heap size. `gc::cell` does not have any virtual functions, and as such does not begin its memory allocation with a V-Table reference. `gc::object` however does start with a V-Table pointer (This is not necessarily a requirement for
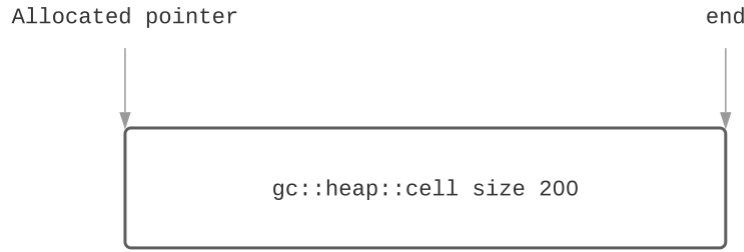
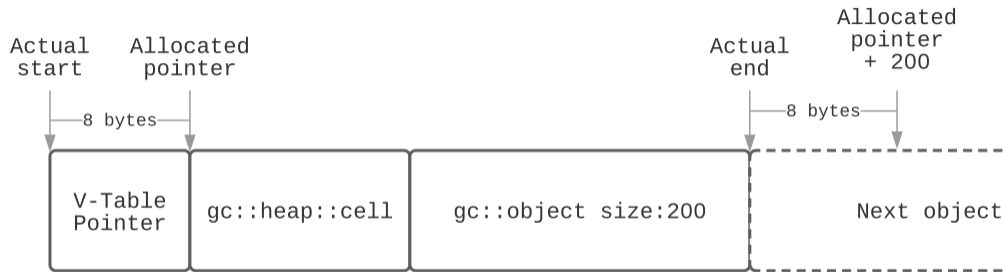Figure 3: Positioning of pointer to free `gc::heap::cell`

Figure 4: Positioning of pointer to `gc::heap::cell` in a `gc::object`

it to function, but its currently kept like this to ensure that users can use virtual functions further down the inheritance tree). The `gc::object` class also inherits from the `gc::cell` class which is the reason for the need to reposition the memory position after allocation. Links in the treadmill list always point to the `gc::heap::cell` object. This is fine initially as the cell begins at the given pointer position.

But this will not work if the cell is actually part of a now garbage collected object. In this second case, the position of the cell object and the position of where that cell actually is are different, due to the additional V-Table pointer (Shown in figure 4).

`malloc` takes this information into account by calling the `gc::heap::cell::actual_position` member function. The contents of this function is set when a `gc::heap::cell` is constructed and gives the data as to whether the cell object is at an offset or not. When a standard cell is created, this function will return the same as the `this` pointer, but if it was constructed during the creation of a `gc::object` it will instead point to `this - sizeof(void*)`. `malloc` must make this adjustment to ensure that any newly created object does not bleed data into the next heap position. `malloc` will also add this cell to the initialization list. This is a blacklist that ensures that objects do not get garbage collected before their initialisation has been finished. Without this safety, a sub-object `malloc` could run out of memory and attempt to garbage collect its parent object's memory. Once the entire object has been fully constructed (an event the system knows has taken place once the `field/static_ptr` constructors havbe been reached), the objects get removed from the initialisation list ready for possible garbage collection.
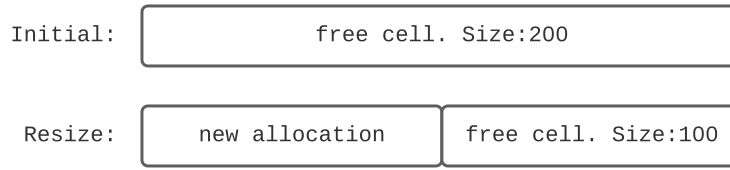
6

Initial: | free cell. Size:200 |

Resize: | new allocation | free cell. Size:100 |

Figure 5: Cell resize after new allocation

Before: | left over 10 | free cell. Size:200 |
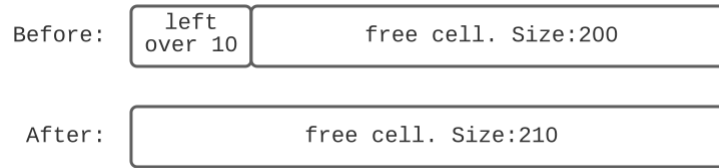
After: | free cell. Size:210 |

Figure 6: Left over memory merging with forward free cell

3.2.2 `gc::heap::heap_struct::attempt_allocate`

This function is responsible principally for finding free memory locations, and where necessary, merging contiguous cells that exist in free memory, `attempt_allocate` attempts to find allocation positions by comparing the new object's size requirements to each cell in the free list, starting from the top of the list (`_free` cell) and working down. If a free cell is found which is large enough, it will break off an amount of memory which is equal to the size of the new cell. If the new object size is the same as the free cell size, this is a very simple process. All that needs to be done is to unlink the free cell from the free list and return the pointer. If however the free cell is larger, a cell resize needs to take place by calling the `gc::heap::cell::resize` function. Cell resizing is performed by moving the start of the free cell object forwards by an amount which is equal to the size of the new object as seen in figure 5.

 `gc::heap::cell` has a certain amount of data that needs to be stored, such as its size and positions in the treadmill and location lists. Because of this, there is a minimum object size for a given cell. If it is found that the space left over in the free cell is too small a cell merge needs to be performed with one of its neighbours. `resize` will first attempt to merge the memory with its forward neighbour if this is not allocated memory. In this case, a new cell is created at the position of the orphan memory, which will incorporate the orphan memory along with the forward cell that it is being merged with.

 If the forward object is allocated memory then it is not possible to merge without corruption of the object. So instead, a merge with the back object (or more specifically our newly allocated object) is performed which will simply act as padding on the end of the object allocation.

 If a free cell is not large enough to accommodate a new object size, `attempt_allocate` will attempt to merge the current free cell with its forward neighbour and try again. If it is unable to merge with its forward neighbour it will move onto the next position in the free list and attempt the same algorithm again. `attempt_allocate` has a loop limit defined when the heap is first created as the `gc::heap::heap_struct::_max_allocation_attempts_before_gc` member. Once the `attempt_allocate` loop has been run this many times, the function will give up and return

7

Before:	new object. Size:70	left over 10

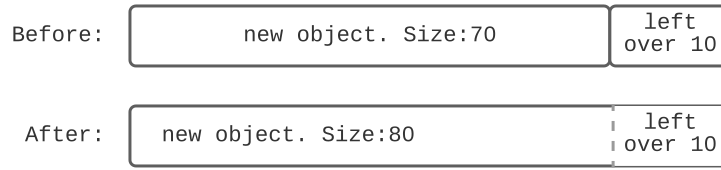| After:	new object. Size:80	left over 10

Figure 7: Left over memory merging with backward, newly allocated object

a `nullptr` in order to allow a garbage collection to take place to free up more memory.

3.2.3 Detecting Free Cells in O(1) Time

Detecting whether or not a cell is currently free or not is important in this system to ensure that data does not get corrupted by cell merges. Because making this check needs to be performed on such a consistent basis, it is alos important that the detection of free cells is a very fast process, in this case, an O(1) time process.

AGC's cell merge system uses the `gc::heap::cell::garunteed_free` function to check to ensure that a cell is definitely in the free list. It is important to note that not every free cell will return true to this function depending on its current state. It only guarantees that no cell that returns true to this function **has** to be free.

To do this, every `gc::heap::cell` has a byte member called `_iteration`. This value corresponds to the byte member `gc::heap::heap_struct::_gc_iteration`. `_gc_iteration` gets iterated every time a garbage collection flip is performed and is used as a notification to all objects that their gc status has been changed.

1. **Use on Objects That are Still Allocated**

   Objects need to keep track of what gc list they are currently in, in a fashion which can be accessed in O(1) time. Objects do this by using the byte member `gc::object::_mark` which can either be `W`, `G` or `B` for *ecru*, *grey* and *black* lists respectively. At different markers, the object will perform different actions when the `gc::cell::gc_mark` member function is called.

   - When in *ecru*, the object moves to *grey* and changes `_mark` to `G`.
   - When in *grey*, the object moves to *black*, calls `gc::cell::gc_mark` on all of its fields, and changes `_mark` to `B`.
   - When in `black`, no action is taken.

   The `_iteration` member becomes important once a gc flip occurs. In this case it will move objects in the *black* list to the *ecru* list, but the objects still display an internal `_mark` of `B` suggesting that it thinks it is in a *black* list, and should therefore do nothing when `gc_mark` is called. Rather than slow down the gc flip by manually setting each object back to `_mark = 'W'`, instead the process is performed lazily by checking for inconsistencies between `gc::heap::cell::_iteration` and `gc::heap::heap_struct::_gc_iteration` during `gc_mark` calls. If when `gc_mark` is called the object sees that its `_iteration` value is behind `_gc_iteration`, it now knowz that a gc flip has occurred since and it needs to reset to `_mark = 'W'`.

8

2. **`_iteration`** in Free Cell Merges

   This same `_iteration` member is also used in determining if a cell can be merged with. Because cells that are currently allocated are required to update their `_iteration` member every gc cycle to keep in sink with the global `_gc_iteration`, this fact can then be exploited in order to tell which cells are not being updated. It can be garunteed that any cell which has an `_iteration` value that is not equal to `_gc_iteration` of (`_gc_iteration - 1`) is actually a free cell.

   - Where `_iteration = _gc_iteration`, the cell must have either recently updated their member, or have been allocated in the current iteration and are therefore not free
   - Cells that have `_iteration = (_gc_iteration - 1)` must either not yet have had the change to update their masrk yet, or have just been freed in the previous cucle.

   If any other `_iteration` value is found, then we must conclude that the cell is out of date and garunteed to be free. Because `_iteration = (_gc_iteration - 1)` can be true on an allocated cell, this means that it only becomes possible to merge with a free cell once two garbage collections have passed since it was first placed in the free list.

3.3   Object Survival and Being Unreferenced

The standard approach for memory management in C++ is to perform reference counting, that is, to count the number of references that are pointing to a given object, and then once no references are left, delete the object. AutoGarbage uses mark and sweep which will instead, as the program is being executed, gradual;ly mark out and reveal the memory cells which are still being referenced. The underlying effect of this method of memory management is that objects tend to stay allocated a little longer than they are atually required to be. This means that the amount of memory used by a program at any time is going to be slightly greater than what is actually require, simply because a gc cycle has yet to occur, or an objects gets sent to the grey/black list just before it gets dereferenced.

   This expanded object lifetime can mean that on occasion, a greater computational overhead is associated with mark and sweep as it can take a couple of cycles for an unreferenced object to finally be freed. Specifically in the AutoGarbage system, it may even take yet another cycle to become a genuinely useful allocatable object, due to the approach taken to ensuring that it is safe to merge with free cells.

A   Other Approaches Explored for Stopping Marking

A few other approaches were explored to perform the task of stopping the marker as it explores an object prior to the magic pointer algorithm that is currently being used within the AutoGarbage program with the goal of reducing the overall memory overhead of objects.

The magic pointer technique (figure 8) requires an 8 byte overhead on 64 bit systems. This compares to a field flag technique (figure 9) which would have added a byte to the top of the field class which could have a different value to a END_GC_FIELDS added member byte at the end of the field list. This approach is better so long as a class has fewer than 7 fields (each field has an extra byte along with the END_GC_FIELD byte makes the 8 bytes taken by the magic pointer technique). A slightly different approach that could be taken is to reserve the top bit of the fields pointer to use in the marking system (figure 10) (set to 0 on fields). This approach would allow for an overhead of only a single byte at the expense of reducing the addressable heap memory from 64bits to 63.

Unfortunately due to the prominence of little endian formatted computers, applying the current system while only observing the most significant byte cannot be used as an optimisation. On big endian machines, a magic number could be used checking only the most significant byte and omitting the rest of the pointer bringing the overhead back to just a single byte without reducing addressable space.
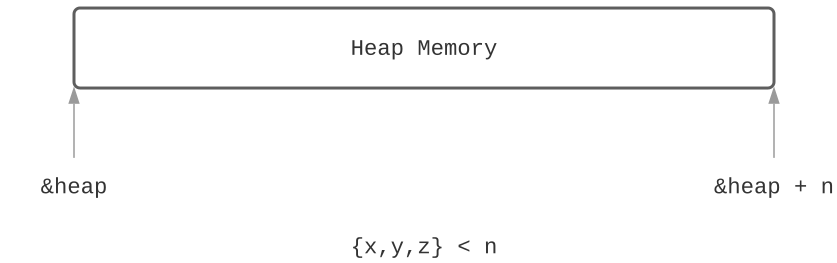
The likely best approach that may be worth moving to in future is the single bit flag within the pointer. This is because although it does reduce addressable space, it only reduces it from a possible ~18 exabytes to ~9 exabytes[A.1], memory spaces that are well beyond even the largest super computers[1]

A.1   Addressable Space

Addresses in C++ direct to individual bytes. The maximum possible unsigned 64 bit number is 18,446,744,073,709,551,614. This therefore is approximately 18 exabytes which is addressable. One bit used as a separate flag, reduces to 2^{63} addresses hence 9 exabytes addressable.

B   Separate Thread Marking for Multi-Threaded Environments

The current AutoGarbage system performs marking of memory when the program makes accesses to objects or when it has run out of memory during an allocation. Another approach that can be taken is to queue up objects to be marked as they get accessed and use a different thread to take objects off of the querue and mark them. This approach decouples the programmers threads from the marking process, ensuring that multiple threads can access the object at the same time without having to worry about performing the correct marking procedure in accordance with the object's current state.

```
┌─────────────────────────────────────────────────────────────────┐
│                                                                   │
│                          Heap Memory                              │
│                                                                   │
└─────────────────────────────────────────────────────────────────┘
     ▲                                                         ▲

  &heap                                                   &heap + n
```

$$\{x, y, z\} < n$$

Object Memory Map

Space	Memory Value	Class Member
8	&heap + x	gc::field<A> a;
8	&heap + y	gc::field<A> a2;
8	nullptr	gc::field<B> b;
8	&heap + z	gc::field<gc::array<A>> arr;
8	&heap - 1	END_GC_FIELDS;
37		

Figure 8: Magic Pointer Technique

```
┌─────────────────────────────────────────────┐
│               gc::field<A>                    │
├─────────────────────────────────────────────┤
│ char _field_flag                              │
│ A* ptr                                        │
└─────────────────────────────────────────────┘
```
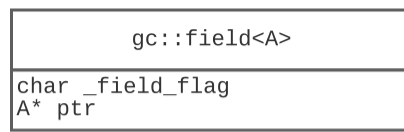
Object Memory Map

Space	Memory Value	Class Member
1	'a'	gc::field<A> a;
8	&heap + x	
1	'a'	gc::field<A> a2;
8	&heap + y	
1	'a'	gc::field<B> b;
8	nullptr	
1	'a'	gc::field<gc::array<A>> arr;
8	&heap + z	
1	'b'	END_GC_FIELDS;
37		

Figure 9: Field Member Flag Technique

$$f(m) = 2^{63} \ \& \ m$$

Object Memory Map

Space	Memory Value	Class Member
8	f(&heap + x)	gc::field<A> a;
8	f(&heap + y)	gc::field<A> a2;
8	f(nullptr)	gc::field<B> b;
8	f(&heap + z)	gc::field<gc::array<A>> arr;
1	0xFFFF	END_GC_FIELDS;
33		

Figure 10: On Pointer Bit Flag Technique

References

[1] 2018 Summit Super Computer at Oak Ridge National Laboratory

2,801,664 GB of volatile memory (4,608 nodes of 512 GB DDR4 + 96 GB HBM2)
`https://www.olcf.ornl.gov/summit/`

[2] Baker, Henry. (1995). The Treadmill: Real-Time Garbage Collection Without Motion Sickness.