C++ Memory Management Innovation: GC Allocator

xushiweizh@gmail.com

2008-4-21

Introduction

Most of the C++ programmers do not benefit from "Garbage Collection" technique (GC). They are sick of deleting objects but have to do this. There are some C/C++ memory GC implementations, but they are complex and are not widely used.

I am going to introduce a new memory management technique named "GC Allocator". "GC Allocator" isn't an implementation, but a concept. Now, we have two "GC Allocator" implementations, named "AutoFreeAlloc" and "ScopeAlloc".

This article consists of three parts:

What is GC Allocator?

1. It is an Allocator for allocating memory.
2. A better Smart Pointer.
3. Creating a GC Allocator and allocating memory should be very fast.
4. Another way of GC.

Concept of GC Allocator

GC Allocator is only a concept. The following is minimum specification for GC Allocator:

```
typedef void (*DestructorType)(void* data);

concept GCAllocator
{
    // Allocate memory without given a cleanup function
    void* allocate(size_t cb);

    // Allocate memory with a cleanup function
    void* allocate(size_t cb, DestructorType fn);

    // Cleanup and deallocate all allocated memory by this GC Allocator
    void clear();

    // Swap two GCAllocator instances
    void swap(GCAllocator& o);
};
```

When you creating a GC Allocator, You can use *STD_NEW*, *STD_NEW_ARRAY* to new objects. Let's see a very simple example:

```
GCAllocator alloc(initArgs); // initArgs depends on implementation

int* intObj = STD_NEW(alloc, int);
int* intObjWithArg = STD_NEW(alloc, int)(10);
int* intArray = STD_NEW_ARRAY(alloc, int, count);

MyObj* obj = STD_NEW(alloc, MyObj);
MyObj* objWithArg = STD_NEW(alloc, MyObj)(100);
```

```
MyObj* objArray = STD_NEW_ARRAY(alloc, MyObj, count);
```

Frankly speaking, I don't like *STD_NEW* and *STD_NEW_ARRAY*. I hope I can use the following syntax:

```
GCAllocator alloc(initArgs);

int* intObj = new(alloc) int;
int* intObjWithArg = new(alloc) int(10);
int* intArray = new(alloc) int[count];

MyObj* obj = new(alloc) MyObj;
MyObj* objWithArg = new(alloc) MyObj(100);
MyObj* objArray = new(alloc) MyObj[count];
```

A Better Smart Pointer

C++ programmers are sick of deleting objects. So they invent a technique named "Smart Pointer". There are many implementations of Smart Pointer. The simplest one is std::auto_ptr. Here is an example:

```
{
    std::auto_ptr<MyObj> obj(new MyObj);
    std::auto_ptr<AnotherObj> obj2(new AnotherObj);
    ... // use obj and obj2 to do something.
}
```

If you don't use Smart Pointer, you have to write the following code:

```
{
    MyObj* obj = new MyObj;
    AnotherObj* obj2;
    try
    {
        obj2 = new AnotherObj;
    }
    catch(...}
    {
        delete obj;
        throw;
    }
    try
    {
        ... // use obj and obj2 to do something.
    }
    catch(...)
    {
        delete obj2;
        delete obj;
        throw;
    }
    delete obj2;
    delete obj;
}
```

When you use a GC Allocator, you can do the same things as the following:

```
{
    GCAllocator alloc(initArgs); // initArgs depends on implementation

    MyObj* obj = STD_NEW(alloc, MyObj);
    AnotherObj* obj2 = STD_NEW(alloc, AnotherObj);
    ... // use obj and obj2 to do something.
}
```

I think that a GC Allocator is a better smart pointer. Why?

First, if you use Smart Pointer technique, when you need an array you have to implement a new a smart pointer type for array object. The following code doesn't work well:

```
{
    std::auto_ptr<MyObj> objArray(new MyObj[count]);
    // ---> Error!!! You can't pass an array pointer to the auto_ptr constructor.

    ... // use objArray to do something.
}
```

But when you use GC Allocator, It is only a piece of cake:

```
{
    GCAllocator alloc(initArgs); // initArgs depends on implementation

    MyObj* objArray = STD_NEW_ARRAY(alloc, MyObj, count);
    ... // use objArray to do something.
}
```

Second, Most of the Smart Pointer implementations (eg. boost::shared_ptr, ATL::CComPtr, etc) are based on "Reference Counting" technique. When an algorithm needs to return a new object, "Reference Counting" is a common solution. For example:

```
boost::shared_ptr<MyObj> algorithm(...)
{
    boost::shared_ptr<MyObj> obj(new MyObj);
    ...
    return obj;
}
```

But "Reference Counting" really isn't a good solution:

1. The Windows COM (based on "Reference Counting") programmers are sick of memory leak endlessly.
2. It's a fact that not all of the C++ programmers like smart pointers, and not all of the C++ programmers like the SAME smart pointer. You have to convert between normal pointers and smart pointers, or between one smart pointer and another smart pointer. Then things become complex and difficult to control.
3. Having a risk of Circular Reference.
4. Tracking down memory leaks is more difficult when an object has a "Reference Count".

When you use GC Allocator, a GC Allocator instance will be passed to the algorithm if it needs to return a new object:

```
template <class AllocT>
MyObj* algorithm(AllocT& alloc, ...)
{
    MyObj* obj = STD_NEW(alloc, MyObj);
    ...
    return obj;
}
```

Note the allocator instance *alloc* is passed as a template class AllocT. At the beginning of this article, I said that GC Allocator was not a implementation, but a concept. Now you know why I say that: most of algorithms don't need to care what the *alloc* instance is.

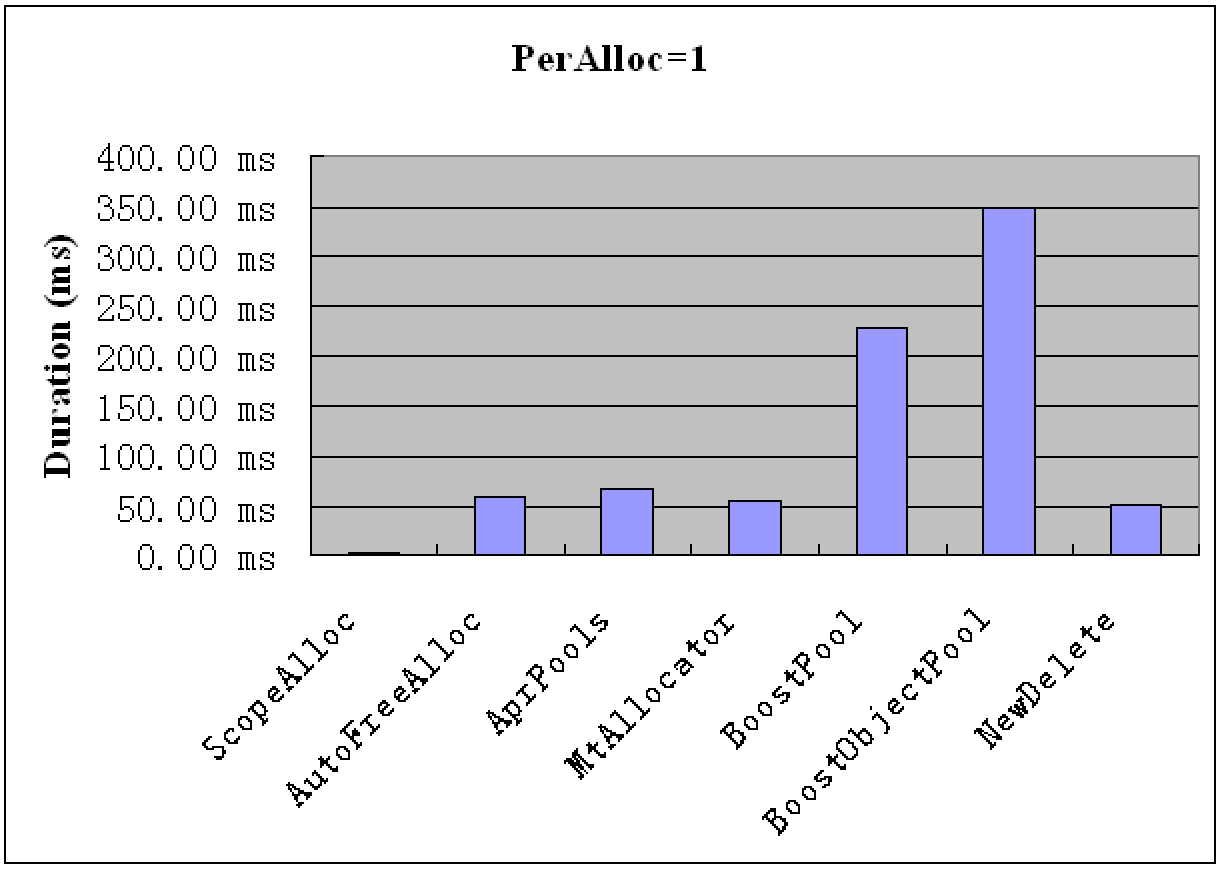
Last, "Smart Pointer" is out, all what you use are just normal pointer. This is very important. It makes the code using "GC Allocator" work together with the code without using "GC Allocator" well.

Creating a GC Allocator and Allocating Memory Should Be Very Fast

Most of allocator implementations optimize allocating a lot of objects. If ONE allocator instance only allocates ONE object instance, they become slower than a normal new/delete allocation. When we consider GC Allocator as Smart Pointer, It should be fast even if It only allocate ONE object instance.

Let's see one of our test result:

PerAlloc	ScopeAlloc	AutoFreeAlloc	AprPools	MtAllocator	BoostPool	BoostObjectPool	NewDelete
1	3.93 ms	59.26 ms	68.58 ms	56.48 ms	227.61 ms	347.08 ms	50.66 ms



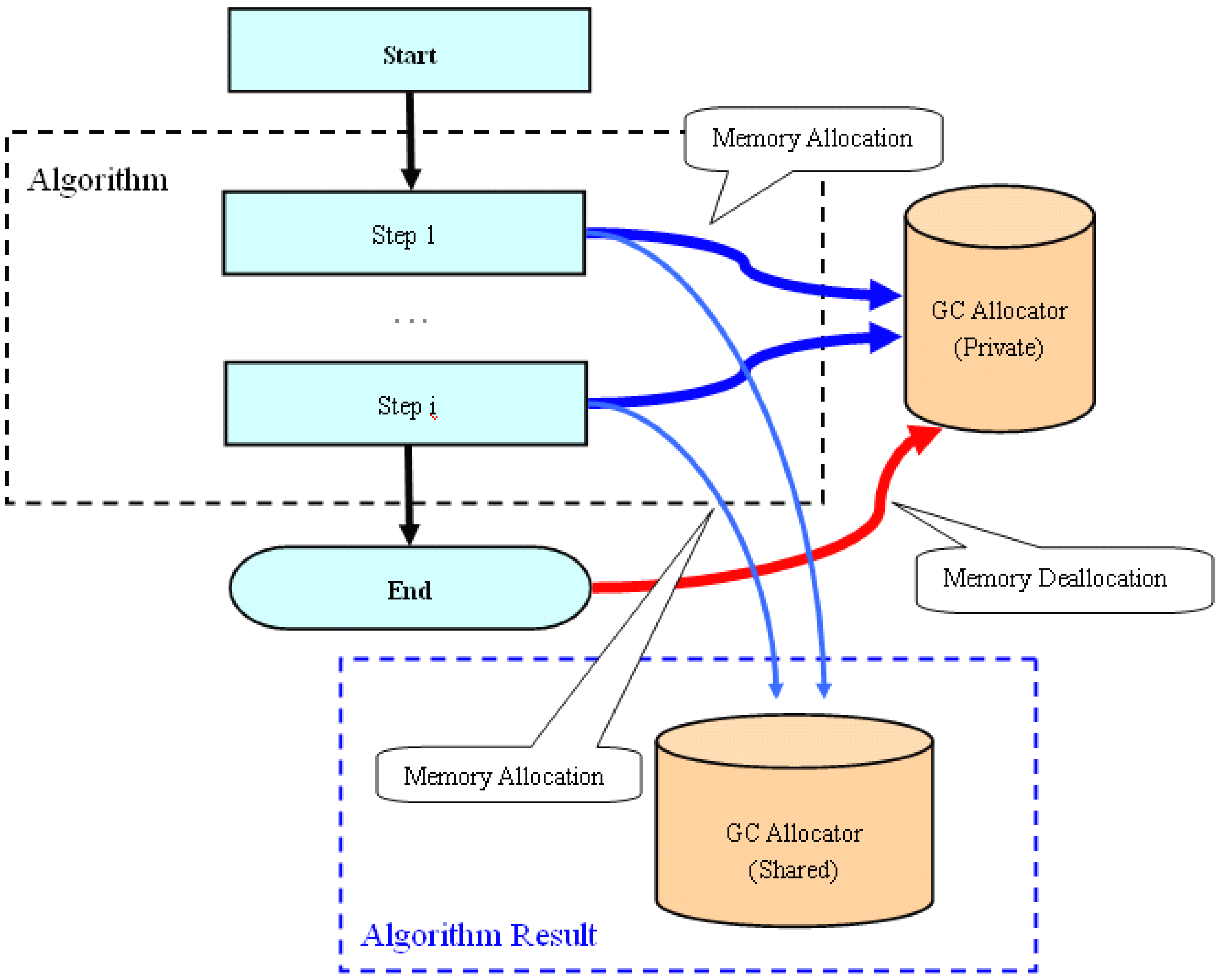
(*PerAlloc* means the number of objects allocated by one allocator instance)

For the detail information about the comparison, see "Allocators Performance Comparison". I 'm excited that ScopeAlloc is observably faster than a normal new/delete allocation and AutoFreeAlloc is close to a normal new/delete allocation.

Another Way of GC

I think that GC Allocator is another way of GC. Of course, GC Allocator does't work as GC in Java or C#. In fact, the core source code of our GC Allocator implementation is only about 100 code lines! It doesn't do too much, but do the most important things.

Generally GC Allocator has an abstract to algorithms like this:



An algorithm may has two GC Allocator instances. One is named "Private GC Allocator". Another is named "Shared GC Allocator". If an object will be returned out, then it will be allocated by "Shared GC Allocator". If an object will be destroyed when the algorithm is end, then it will be allocated by "Private GC Allocator". The pseudo code looks like this:

```
ResultDOM* algorithm(GCAllocator& sharedAlloc, InputArgs args)
{
    GCAllocator privateAlloc(sharedAlloc);
    ...
    ResultDOM* result = STD_NEW(sharedAlloc, ResultDOM);
    ResultNode* node = STD_NEW(sharedAlloc, ResultNode);
    result->addNode(node);
    ...
    TempVariable1* temp1 = STD_NEW(privateAlloc, TempVariable1);
    TempVariable2* temp2 = STD_NEW(privateAlloc, TempVariable2);
    ...
    return result;
}
```

The Private GC Allocator (named *privateAlloc*) works like a "Smart Pointer". But unlike "Smart Pointer", ONE GC Allocator instance manages a group of objects, not ONE BY ONE.

If the amount of private allocated objects is small, *privateAlloc* is not needed. Then the algorithm become like this:

```
ResultDOM* algorithm(GCAllocator& alloc, InputArgs args)
{
    ResultDOM* result = STD_NEW(alloc, ResultDOM);
    ResultNode* node = STD_NEW(alloc, ResultNode);
    result->addNode(node);
    ...
    TempVariable1* temp1 = STD_NEW(alloc, TempVariable1);
    TempVariable2* temp2 = STD_NEW(alloc, TempVariable2);
    ...
    return result;
}
```

In any case, you don't need to delete objects manually. This is why I call GCAllocator "GC Allocator".

GC Allocator Implementations: ScopeAlloc and AutoFreeAlloc

We have two "GC Allocator" implementations, named "AutoFreeAlloc" and "ScopeAlloc". A brief summary of this section follows:

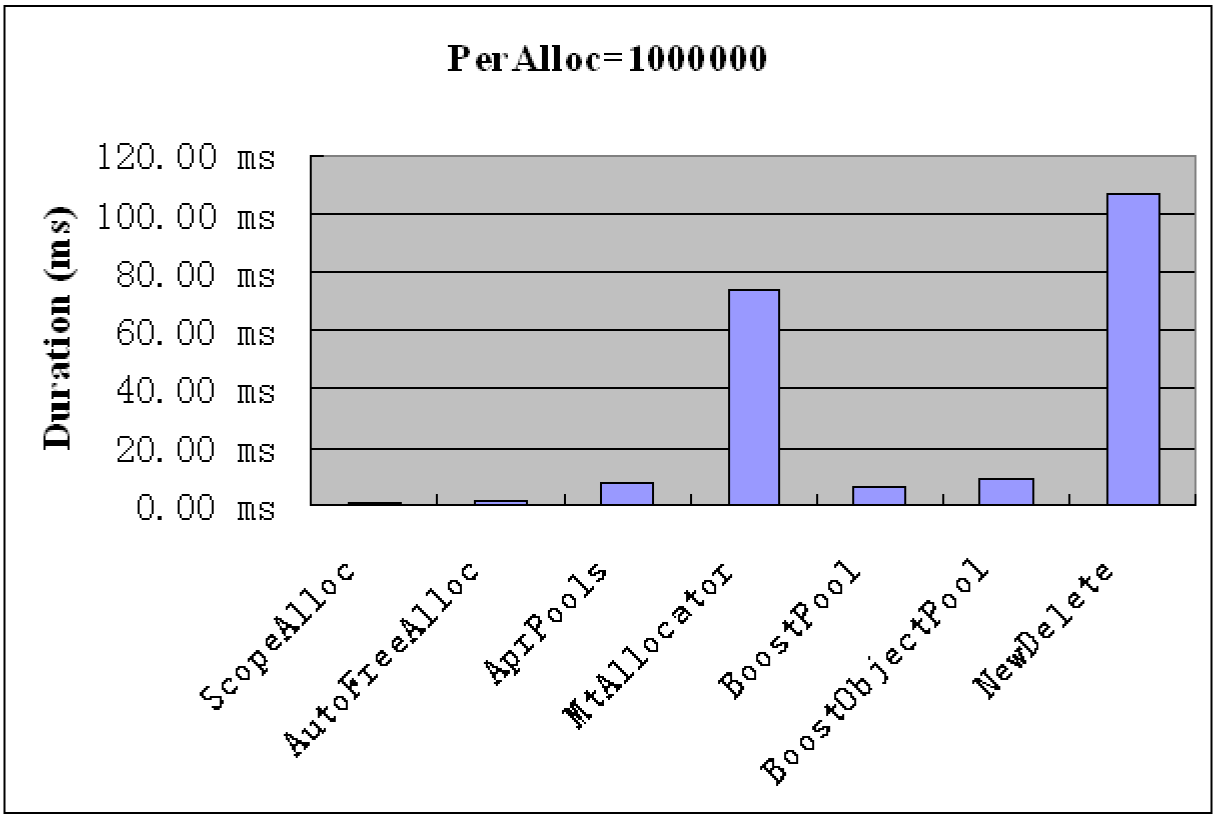
1. Performance: they are faster than all other allocators you ever seen.
2. The infrastructure of ScopeAlloc and AutoFreeAlloc.
3. A huge stack.
4. Only about 100 core code lines.
5. No multithreaded locks (no need).
6. When to use AutoFreeAlloc.
7. Open source.

Faster Than All Allocators You Ever Seen

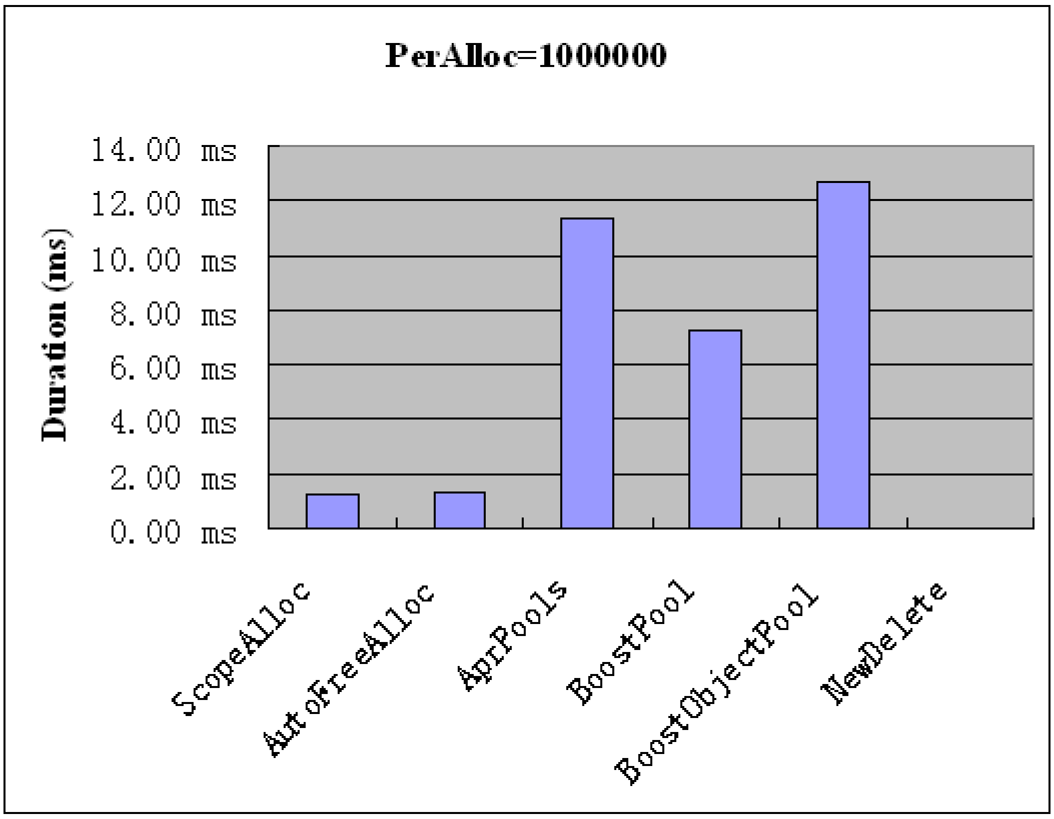
We toke performance comparison of:

- AutoFreeAlloc
- ScopeAlloc
- APR Pools (Apache Portable Runtime)
- MT Allocator (GNU C++)
- Boost Pool (boost::pool)
- Boost ObjectPool (boost::object_pool)
- NewDelete (new/delete)

In linux platform, we got the following result:



In windows platform, we got the following result (I remove the NewDelete bar because it is too slow):



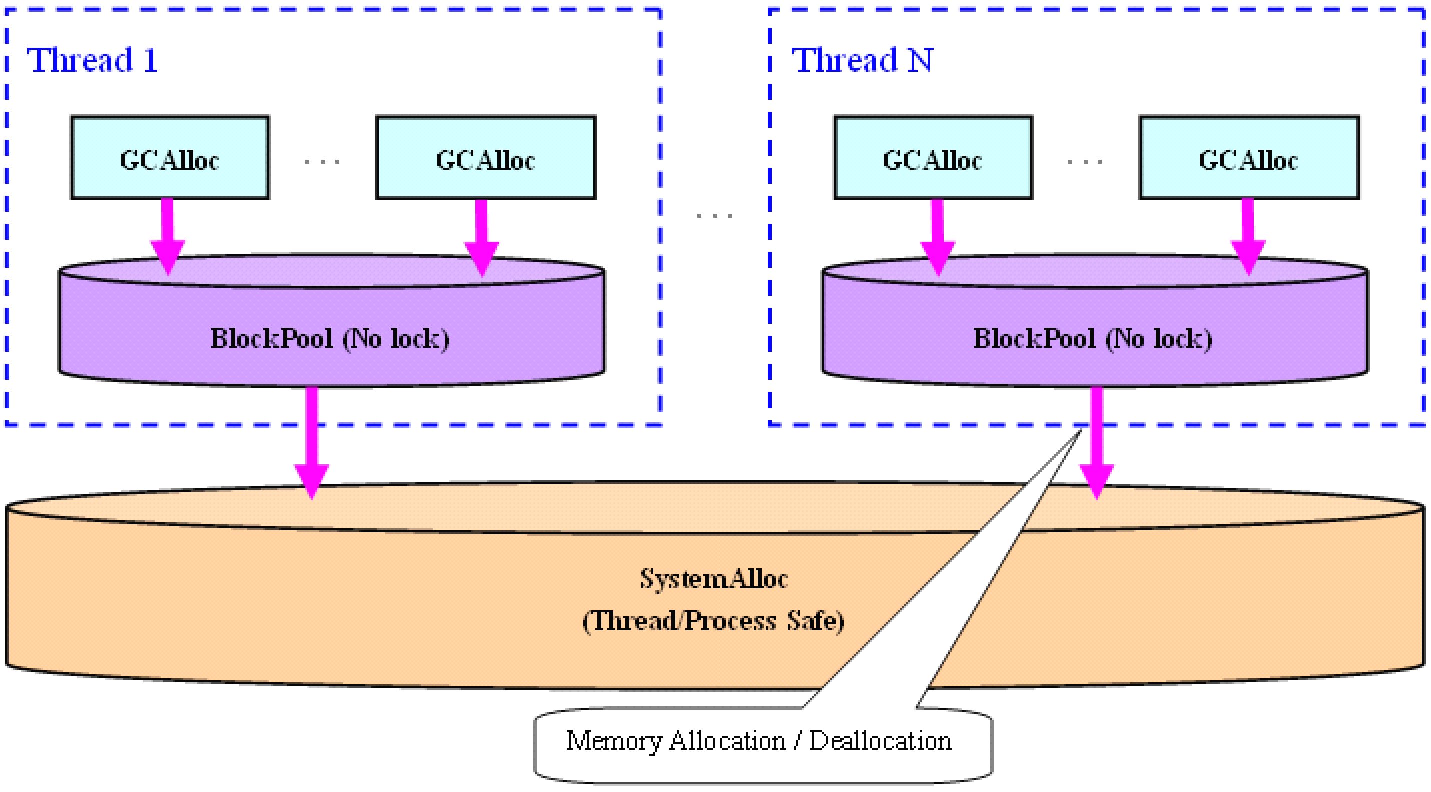
For the detail information about the comparison, see "Allocators Performance Comparison".

Why ScopeAlloc and AutoFreeAlloc are so fast? They benefit from:

1. Its good allocation algorithm.
2. No multithreaded locks.
3. C++ inline functions.

The Infrastructure of ScopeAlloc and AutoFreeAlloc

The infrastructure of ScopeAlloc and AutoFreeAlloc follows:



There are three basic concepts: SystemAlloc, BlockPool and GCAlloc.

SystemAlloc is an abstract of the underlying system memory management service. It is only a wrapper of malloc/free procedures of C runtime:

```
class SystemAlloc
{
public:
    void* allocate(size_t cb) { return malloc(cb); }
    void deallocate(void* p)    { free(p); }
    void swap(SystemAlloc& o) {}
};
```

BlockPool is a Memory Pool. It works as a cache pool to accelerate the speed of allocating memory. BlockPool provides the same interface as SystemAlloc:

```
class BlockPool
{
public:
    BlockPool(int cacheSize = INT_MAX);

    void* allocate(size_t cb);
    void deallocate(void* p);
    void swap(SystemAlloc& o);
};
```

Why BlockPool can accelerate the speed of allocating memory? It isn't because BlockPool is a memory pool (you know SystemAlloc may also be implemented by using Memory Pool technique), but BlockPool doesn't need multithreaded locks.

The last concept is GCAlloc. It is the core of GC Allocator. And it is only about 100 core code lines!

GCAlloc: A Huge Stack

The implementation class of GCAlloc is named "GCAllocT". It provides the following interface:

```
template <class _Alloc>
class GCAllocT
{
public:
    GCAllocT();
    explicit GCAllocT(const _Alloc& alloc);
    explicit GCAllocT(GCAllocT& owner);

    // Allocate memory without given a cleanup function
    void* allocate(size_t cb);

    // Allocate memory with a cleanup function
    void* allocate(size_t cb, DestructorType fn);

    // Cleanup and deallocate all allocated memory by this GC Allocator
    void clear();

    // Swap two GCAllocator instances
    void swap(GCAllocT& o);
};

typedef GCAllocT<SystemAlloc> AutoFreeAlloc;
typedef GCAllocT<ProxyBlockPool> ScopeAlloc;
```

AutoFreeAlloc and ScopeAlloc are both based on GCAllocT. The only different thing is that AutoFreeAlloc is based on SystemAlloc (global malloc/free allocation procedures), while ScopeAlloc is based on a BlockPool (a cache allocator to accelerate the speed of memory allocation).

AutoFreeAlloc and ScopeAlloc have distinct performance difference due to this difference.

> ScopeAlloc has the best performance in any condition. And if we allocate enough objects, performance of AutoFreeAlloc is close to ScopeAlloc.

For the detail information, see "Allocators Performance Comparison".

You know, the fastest "Allocator" is "Stack". C/C++ allocate auto objects (also named "Stack Objects") on "Stack". But "Stack" has many of limit:

1. When exiting a procedure, all "Stack Objects" will be destroyed. So, you can't return a "Stack Object".
2. You can't allocate too many "Stack Objects", because "Stack" has limited size.

The basic idea of GCAlloc is implemented as "a huge stack" on heap. It have similar performance as "Stack". I don't explain the detail implementation here. If you want to dive into it, refer the source code.

No Multithreaded Locks

Why doesn't GC Allocator need multithreaded locks?

Memory allocation = System memory block management + Memory management algorithm

The underlying system memory block management is provided by OS. It will optimize large memory block allocation (allocating small objects is supported, but doesn't need to optimize). And It is thread/process safe.

Memory management algorithm is provided by C/C++ runtime library, or other libraries. Memory management algorithms ARE only algorithms. Most of them are designed to be thread safe.

If we use global new/delete or malloc/free procedures to allocate memory, thread safe is a MUST (because all threads use these procedures), not OPTIONAL. But if we use allocator instances to manage memory, then thread safe becomes OPTIONAL.

Why? Sharing GC Allocator in multi threads is not recommended. It means that sharing memory between threads is also not recommended. If you REALLY want to share memory, use new/delete or anything else.

For users of GC Allocator, we suggest that only use ONE BlockPool instance in ONE thread, and ONE thread may use multiple PRIVATE "ScopeAlloc" instances (depend on your requirement) to allocate memory.

And this makes ScopeAlloc be the fastest allocator!

When to use AutoFreeAlloc

You know, ScopeAlloc is faster than AutoFreeAlloc in any condition. Then you may wonder when to use AutoFreeAlloc. Here are some conditions that you can consider to use AutoFreeAlloc:

1. An algorithm that don't want to accept a BlockPool or ScopeAlloc parameter. If we use ScopeAlloc, we need a BlockPool or ScopeAlloc instance to construct a new ScopeAlloc object. But in some case, we don't want our users to know the implementation detail of using ScopeAlloc.
2. An algorithm that need a lot of memory allocations. If we allocate enough objects, performance of AutoFreeAlloc is close to ScopeAlloc (see "Allocators Performance Comparison").
3. An algorithm that will release all allocated memory when the algorithm is end.

Open Source

Yes, ScopeAlloc/AutoFreeAlloc is open source. And it's licensed under Common Public License(CPL). you can find more information in http://code.google.com/p/stdext/.

Here are quick links for source code of ScopeAlloc/AutoFreeAlloc:

- ✧ AutoFreeAlloc.h - class GCAllocT, AutoFreeAlloc
- ✧ ScopeAlloc.h - class BlockPool, ScopeAlloc
- ✧ SystemAlloc.h - class SystemAlloc

Applications based on GC Allocator

Is GC Allocator useful? Yes. Things are changed for C++ developers! We also can benefit from GC like Java and C#! And There are already some applications based on it. Here are part of them:

A Word File Writer

I wrote a word file format writer with GC Allocator. I was excited that It is the fastest word file writer component I ever seen.

Interested in it? See The Fastest Word File Writer.

Rope based on GC Allocator

Rope is a complex string implementation with scaled performance. The original rope implementation is appeared in SGI STL. I rewrite the rope class based on GC Allocator. Code size is much reduced and performance is better.

Interested in it? See Rope on GC Allocator.

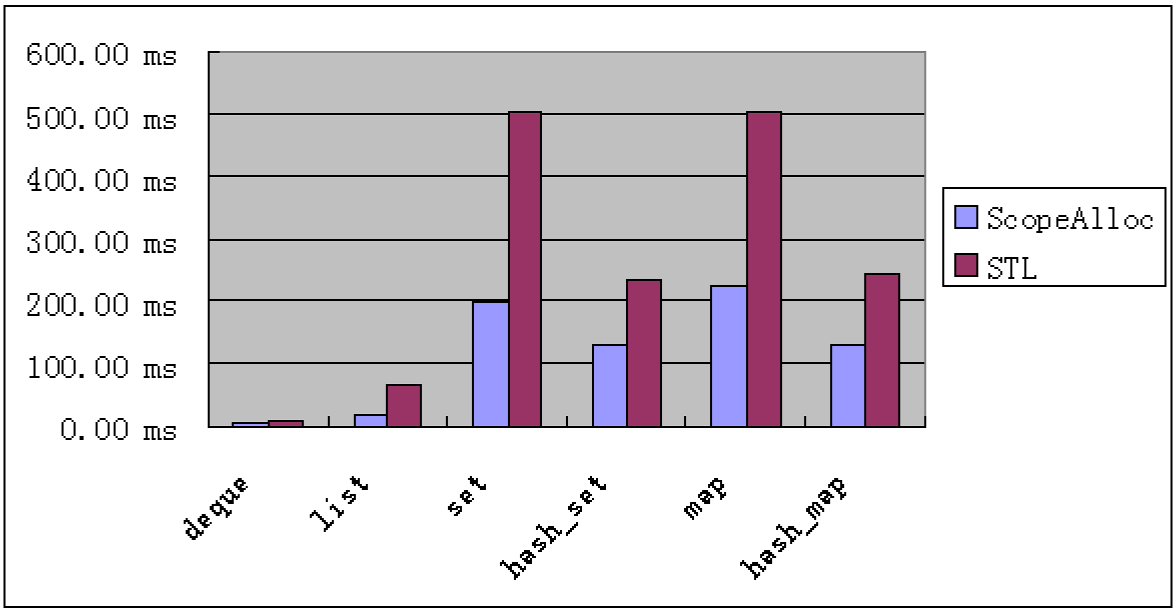
STL Containers based on GC Allocator

Not only rope, but most of all STL containers can be based on GC Allocator, including:

- Deque
- List, Slist
- Map, MultiMap
- Set, MultiSet
- HashMap, HashMultiMap
- HashSet, HashMultiSet

When we use ScopeAlloc, performance of STL containers has distinct promotion. Here is one of our test result:

	deque	list	set	hash_set	map	hash_map
ScopeAlloc	5.71 ms	20.32 ms	198.44 ms	129.68 ms	225.12 ms	130.80 ms
STL	8.34 ms	66.56 ms	504.81 ms	232.63 ms	505.34 ms	242.21 ms



For the detail information about the comparison, see "Allocators Performance on STL Collections".

Note that we don't provide all STL containers based on GC Allocator. STL containers of linear data structure (eg. std::vector, std::basic_string/std::string, etc) don't need a GC allocator.

Related Topics

- Allocators Performance Comparison
- Allocators Performance on STL Collections
- Rope based on GC Allocator
- The Fastest Word File Writer