Writing a Pool Allocator

In the previous article on Writing a Memory Allocator we discussed and implemented a *generic* memory allocator. We have seen how a memory is requested from OS (through the *memory mapping*), and in particular focused on different strategies of the **Free-list** allocation.

In today’s lecture we’ll be discussing a *Pool allocator*.

Overview

A **Pool allocator** (or simply, a **Memory pool**) is a variation of the fast *Bump-allocator*, which in general allows O(1) allocation, when a free block is found *right away*, without searching a free-list.

To achieve this fast allocation, usually a pool allocator uses blocks of a *predefined size*. The idea is similar to the Segregated list, however with even faster block determination.

The approach can greatly speed up performance in systems which work with many objects of predefined shapes. For example, in a gaming app we may need to allocate *hundreds*, if not *thousands* of objects of the same type. In this case a fragmented malloc might be a source of a slower allocation. That’s why gaming consoles actively involve memory pools for such objects.

So, let’s dive into the details, and implement one.

Blocks and Chunks

A pool allocator operates with concepts of **Blocks (Pools)**, and **Chunks** within each block.

Each chunk is of *predefined size*, and encodes the **Object header**, which stores a *meta-information* needed for Allocator’s or Collector’s purposes.

Let’s start from the chunks first.

Chunks: individual objects

Since the size is predefined, we don’t need to store it in the header, and only can keep a reference to the *next* object. We represent an allocation as a Chunk structure:

```
struct Chunk {
  /**
   * When a chunk is free, the `next` contains
the
   * address of the next chunk in a list.
   *
   * When it's allocated, this space is used by
   * the user.
   */
  Chunk *next;
};
```

When a chunk is allocated, Mutator (the user code) can *fully* occupy it, including the space initially taken by our next pointer. For this reason, we don’t even need a flag whether a chunk is free — this is solved by the **Allocation pointer** which always points to a *current free chunk*, as we will see shortly.

Here is the picture how the free chunks look in memory:

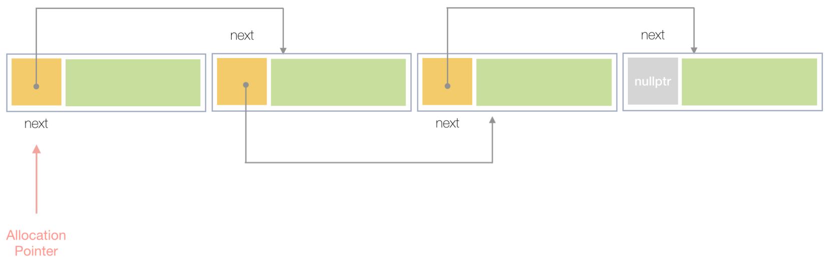

Figure 1. Free chunks

So, — a simple linked list of all available free chunks. Notice again how the allocation pointer is set to the current free chunk, which will be found right away on allocation request.

And once some objects are allocated, the allocation pointer is *advanced* accordingly, and still points the current free chunk, ready to be returned right away.

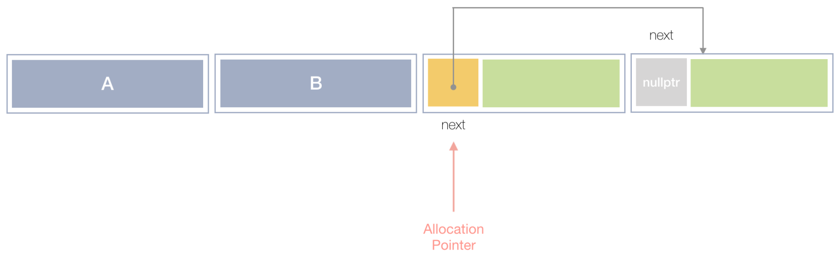

Figure 2. Allocated blocks

Blocks: groups of chunks

To support this fast allocation, the memory for chunks should already be *preallocated*. This preallocation is exactly known as a *pool of objects*, and which we call as a **Block** in our implementation.

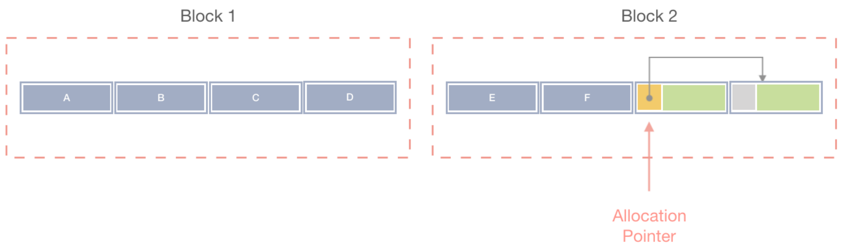

Figure 3. Blocks: groups of chunks

A block though is not represented as an actual separate structure in our code. It’s rather an abstract concept, which *groups* the chunks by just allocating *enough space* to store the needed amount of chunks.

The size of a block is determined by the *number of chunks* per block.

Let’s start defining our PoolAllocator class, accepting the number of chunks as a parameter:

```
/**
```

```
 * The allocator class.
 *
 * Features:
 *
 *   - Parametrized by number of chunks per block
 *   - Keeps track of the allocation pointer
 *   - Bump-allocates chunks
 *   - Requests a new larger block when needed
 *
 */
class PoolAllocator {
 public:
  PoolAllocator(size_t chunksPerBlock)
    : mChunksPerBlock(chunksPerBlock) {}

  void *allocate(size_t size);
  void deallocate(void *ptr, size_t size);

 private:
  /**
   * Number of chunks per larger block.
   */
  size_t mChunksPerBlock;

  /**
   * Allocation pointer.
   */
  Chunk *mAlloc = nullptr;

  /**
   * Allocates a larger block (pool) for chunks.
   */
  Chunk *allocateBlock();
};
```
As we see, the class keeps track of the *allocation pointer* (mAlloc), has private routine to allocateBlock when a new block is needed, and also provides the standard allocate, and deallocate methods as public API.

Let’s focus on the allocation first.

Allocation

To satisfy an allocation request, we need to return a pointer to a free chunk within a current block. However, when there are no chunks left in a current a block, or when we don’t have any block yet at all, we need to allocate a *block itself* first — via the standard malloc mechanism. The sign for this is when the allocation pointer mAlloc is set to nullptr.

```
void *PoolAllocator::allocate(size_t size) {

  // No chunks left in the current block, or no any block
  // exists yet. Allocate a new one, passing the chunk size:

  if (mAlloc == nullptr) {
    mAlloc = allocateBlock(size);
  }
  ...
}
```

Now let’s see what’s happening in the allocateBlock.

Block allocation

The size of the large block is the number of chunks per block, multiplied by the chunk size. Once the block is allocated, we also need to *chain* all chunks in it, so we can easily access the next pointer in each chunk.

```
/**
 * Allocates a new block from OS.
 *
 * Returns a Chunk pointer set to the beginning of the block.
 */
Chunk *PoolAllocator::allocateBlock(size_t chunkSize) {
  cout << "\nAllocating block (" << mChunksPerBlock << " chunks):\n\n";

  size_t blockSize = mChunksPerBlock * chunkSize;

  // The first chunk of the new block.
  Chunk *blockBegin = reinterpret_cast<Chunk *>
(malloc(blockSize));

  // Once the block is allocated, we need to
chain all
  // the chunks in this block:

  Chunk *chunk = blockBegin;

  for (int i = 0; i < mChunksPerBlock - 1; ++i) {
    chunk->next =
        reinterpret_cast<Chunk *>
(reinterpret_cast<char *>(chunk) + chunkSize);
    chunk = chunk->next;
  }

  chunk->next = nullptr;

  return blockBegin;
}
```
As a result we return the Chunk pointer to the beginning of the block — the blockBegin, and this value is set to the mAlloc in the allocate function.

Now let's get back to the allocate, and handle the chunks allocation within a block.

Chunk allocation

OK, so we have allocated a block, and now mAlloc is not nullptr. In this case we return a free chunk at the current position of the allocation pointer mAlloc. We also advance (bump) the allocation pointer further for the future allocation requests.

Let's see at the full allocate function implementation:

```
/**
 * Returns the first free chunk in the block.
 *
 * If there are no chunks left in the block,
 * allocates a new block.
 */
void *PoolAllocator::allocate(size_t size) {

  // No chunks left in the current block, or no any block
  // exists yet. Allocate a new one, passing the chunk size:

  if (mAlloc == nullptr) {
    mAlloc = allocateBlock(size);
  }

  // The return value is the current position of
  // the allocation pointer:

  Chunk *freeChunk = mAlloc;

  // Advance (bump) the allocation pointer to the next chunk.
  //
  // When no chunks left, the `mAlloc` will be set to `nullptr`, and
  // this will cause allocation of a new block on the next request:

  mAlloc = mAlloc->next;

  return freeChunk;
}
```

OK, we can now Bump-allocate chunks within a block, and malloc-allocate the blocks from OS. Now let’s see at the deallocation.

Deallocation

Deallocating a chunk is simpler — we just return it at the *front* of the chunks list, setting the mAlloc pointing to it.

Here is the full code of the deallocate function:

```
/**
 * Puts the chunk into the front of the chunks
list.
 */
void PoolAllocator::deallocate(void *chunk,
size_t size) {

  // The freed chunk's next pointer points to the
  // current allocation pointer:

  reinterpret_cast<Chunk *>(chunk)->next =
mAlloc;

  // And the allocation pointer is now set
  // to the returned (free) chunk:

  mAlloc = reinterpret_cast<Chunk *>(chunk);
}
```

The next figure shows how the allocation pointer is adjusted after the block A is freed, and also how the next pointer of the returned block A points now to the previous position of the mAlloc:

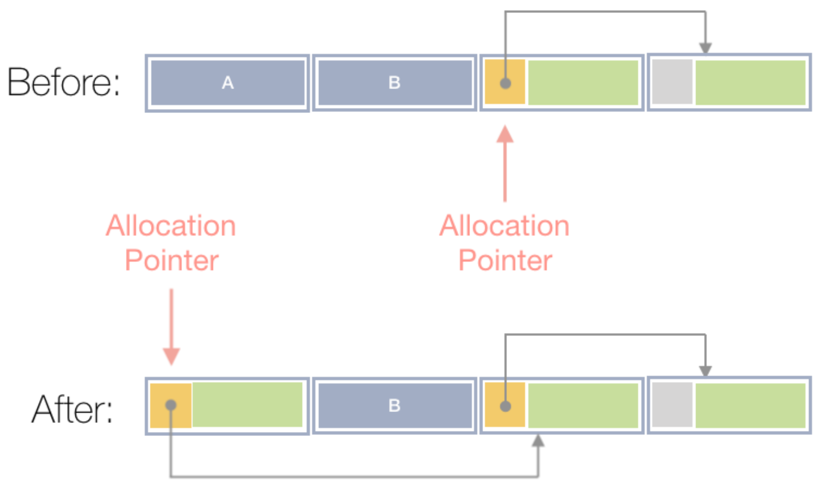

Figure 4. Chunk deallocation

Now when we can allocate and deallocate, let’s create a class with our custom pool allocator, and see it in action.

Objects with custom allocator

C++ allows overriding default behavior of the new and delete operators. We use this advantage to setup our pool allocator, which will be handling the allocation requests.

```cpp
/**
 * The `Object` structure uses custom allocator,
 * overloading `new`, and `delete` operators.
 */
struct Object {

  // Object data, 16 bytes:

  uint64_t data[2];

  // Declare out custom allocator for
  // the `Object` structure:

  static PoolAllocator allocator;

  static void *operator new(size_t size) {
    return allocator.allocate(size);
  }

  static void operator delete(void *ptr, size_t size) {
    return allocator.deallocate(ptr, size);
  }
};

// Instantiate our allocator, using 8 chunks per block:

PoolAllocator Object::allocator{8};
```

Once the allocator is declared and instantiated for our Object class, we can now normally create instances of the Object, and the allocation requests should be routed to our pool allocator.

Usage and testing

Finally, let’s test our allocator, and see the management of the blocks and chunks in action.

```
#include <iostream>

using std::cout;
using std::endl;

int main(int argc, char const *argv[]) {

  // Allocate 10 pointers to our `Object` instances:

  constexpr int arraySize = 10;

  Object *objects[arraySize];

  // Two `uint64_t`, 16 bytes.
  cout << "size(Object) = " << sizeof(Object) << endl << endl;

  // Allocate 10 objects. This causes allocating two larger,
  // blocks since we store only 8 chunks per block:

  cout << "About to allocate " << arraySize << " objects" << endl;

  for (int i = 0; i < arraySize; ++i) {
    objects[i] = new Object();
    cout << "new [" << i << "] = " << objects[i] << endl;
  }

  cout << endl;

  // Deallocated all the objects:

```

```
  for (int i = arraySize; i >= 0; --i) {
    cout << "delete [" << i << "] = " << objects[i] << endl;
    delete objects[i];
  }

  cout << endl;

  // New object reuses previous block:

  objects[0] = new Object();
  cout << "new [0] = " << objects[0] << endl << endl;
}
```

As a result of this execution you should see the following output:

```
size(Object) = 16

About to allocate 10 objects

Allocating block (8 chunks):

new [0] = 0x7fb266402ae0
new [1] = 0x7fb266402af0
new [2] = 0x7fb266402b00
new [3] = 0x7fb266402b10
new [4] = 0x7fb266402b20
new [5] = 0x7fb266402b30
new [6] = 0x7fb266402b40
new [7] = 0x7fb266402b50

Allocating block (8 chunks):

new [8] = 0x7fb266402b60
new [9] = 0x7fb266402b70

delete [9] = 0x7fb266402b70
delete [8] = 0x7fb266402b60
delete [7] = 0x7fb266402b50
delete [6] = 0x7fb266402b40
delete [5] = 0x7fb266402b30
delete [4] = 0x7fb266402b20
delete [3] = 0x7fb266402b10
delete [2] = 0x7fb266402b00
delete [1] = 0x7fb266402af0
delete [0] = 0x7fb266402ae0

new [0] = 0x7fb266402ae0
```

Notice, that specific addresses may be different on your machine, however what important here, is that the objects (chunks) in a block are dense-allocated with a Bump-allocator, following one after another.

Also notice how we started reusing the address 0x7fb266402ae0 when all objects were deallocated, and we received a new allocation request.

Summary

A pool allocator is a useful and practical tool, which you can use to speed up your app in case of many *predefined size* objects. As an exercise, experiment and extend the allocator with some other convenient methods: for example, allow deallocating a whole block *at once* if requested, and returning it back to the global OS allocator.

You can find a full source code for this article in this gist.

You can also enroll to full course here:

Let's do a quick recap of the **Pool allocator** topic:

- It is used when we need to fast-allocate *a lot* of objects with *predefined size*
- Segregates heap by **Blocks** and **Chunks**
- Uses **Bump-allocation** for chunks within a block
- Allocates a *new block* on-demand when no chunks are left
- Fast deallocation: *prepends* a chunk at the *front* of the list

- Reuses freed chunks on future allocations
- Can deallocate a *whole block* at once if needed

I hope you enjoyed this lecture and found it useful! As always I’ll be glad to answer any questions in comments.

Further reading

Written by: Dmitry Soshnikov
Published on: Oct 12th, 2019