Pool Allocator

Introduction

A memory allocator, broadly speaking, is a system that is responsible for managing a segment of memory. This responsibility involves satisfying requests for memory, as well as keeping track of used and unused portions of the segment. Allocator designs vary greatly, and ultimately depend on the usage patterns and constraints of the context in which they are used. Memory with a LIFO structure, for example, only permits deallocations in reverse allocation order, which demands very little overhead and thus generally allows for fast allocations and deallocations. An example of this is the "stack" in C++ application memory [1].

At the opposite end, a *general purpose allocator* manages memory with little or no structure at all. These allocators must be able to handle allocations of practically any size and with no predetermined lifetime – a flexibility that requires significant overhead to satisfy. This is one contributing factor to why allocations on the "heap", i.e. the unstructured segment of a C++ application, are generally slower in comparison to stack allocations [2].

Custom allocators are a (fairly broad) family of strategies intended to improve memory performance in situations with specific usage patterns and constraints. A *Linear Allocator*, for example, does not support deallocations and must clear all allocated memory at once. This may be useful for per-frame data in a computer game. A *Stack Based Allocator* may allow scope-like behavior where memory is allocated and then deallocated from within a subroutine.

The allocator you will implement in this assignment is a variant of a **Pool Allocator**. This type of allocator allocates and manages a chunk of memory that is (implicitly) divided into equally sized elements. Any allocation smaller than or equal to the element size is permitted, and deallocations may take place in any order. Unused elements are kept track of using a *free-list*. Unlike `std::vector`, Pool Allocators typically facilitates *sparse* rather than *dense* storage – meaning that unused elements are interspersed with used elements. This type of allocator is less flexible than default allocators such as `new` and `delete`, but is potentially much faster and can be used for a wide range of memory needs in a game engine.

Design

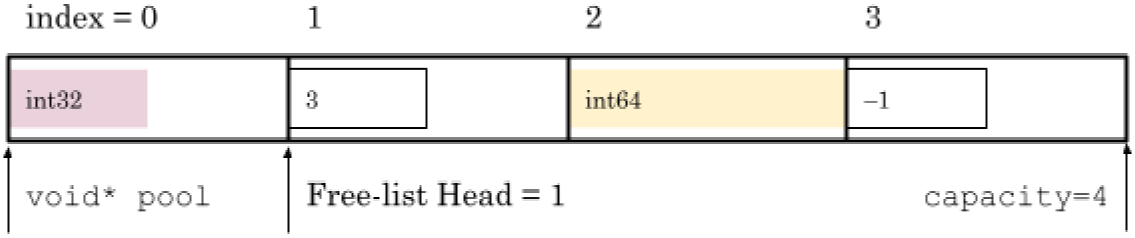

Figure 1: Example layout for our Pool Allocator. The pool itself is a raw allocation with capacity for 4x 8-bytes. Here, elements 0 and 2 are *used* and contain an `int32_t` and an `int64_t`, respectively.
Elements 1 and 3 are *unused* and part of the free-list. The index of the first free element (1) is stored by the free-list *head*. Indices are represented by a 4-byte unsigned type.

These are the main design points for the Pool Allocator in this assignment:

- The pool allocates **raw memory** and keeps it throughout its lifetime
- The pool is defined with a **fixed element size**, `ElementSize`.
- The pool keeps track of used and unused elements using a free-list, implemented as an embedded singly linked-list. See Figure 1.
- Objects of any type and size can be constructed (i.e. allocated and initialized) by the pool, as long as the object size is ≤ `ElementSize`.
- When an object is deallocated, the now-vacant element is returned to the free-list.
- (Upon request, the pool counts and returns the number of free elements.)
- (Upon request, the pool dumps its content to `std::cout`.)

Figure 1 provides a snapshot of a pool with some content in it. Note how the free-list uses the space of unused elements. Also note that objects smaller than `ElementSize` have gaps before the beginning of the next element.

Specification

Here is a detailed list of what is needed. Some points are elaborated upon further down.

Mandatory	
Pool definition	Takes `ElementSize` as a template argument.
Pool creation	Takes `capacity` = max number elements of as an argument. *Actions* ● Allocates **raw memory** to accommodate `capacity` number of elements of size `ElementSize`. *Example* `DynamicPool<8> pool {16};`

Object allocation & initialization	**Variadic template function** `create(...)`, taking a *type* (the type to create) and a *variable* number of template arguments (to construct the object).
	Actions
	● At a free element, constructs the requested type from the provided arguments using **forwarding** and **placement new**.
	● Removes the element from the free-list.
	Checks
	● There pool may not be full (see *policy* below)
	● The requested type must fit within an element.
	Example
	`int* myInt = pool.create<int>(1)`
	`A* myA = pool.create<A>(1, 2, 3)`

Object deallocation	Member function `destroy(T* ptr)`, where `T` is a template parameter. *Actions* ● Calls the destructor of the object. ● Adds the element to the free-list.

	Checks
	The address must be inside the pool's memory range.
	
	Example
	`pool.destroy(myInt)`
	`pool.destroy(myA)`

Policy when about to exceed max capacity	Throws `bad_alloc`. *Alternatives* ● Have support for multiple chunks and add chunks when needed (multi-pool). ● **Reallocate** to a larger memory chunk (similar to `std::vector`). Invalidates existing pointers, unless *indices* are used rather than raw pointers.
Not mandatory	

Free count
 	
Debug print content
	`int count_free()`
	Returns the number of free elements
	
	`void dump_pool()`
	Returns a string representation of the current content.
	
	These operations should be implemented using the **visitor pattern**. The pool should have a private member function `freelist_visitor(...)`, that calls a custom function for each free index. The custom function is received as a template function.
	
	Example usage (count free elements):
	
	`index_type count_free() const`
	`{`
	`  int index_count = 0;`
	`  const auto index_counter =`
	`    [&](index_type i) {`
	`      index_count++;`
	`    };`
	`  freelist_visitor(index_counter);`
	
	`  return index_count;`
	`}`

Grading

Grading is by means of presenting your solution to a teacher.
Your solution should adhere to the Design and Specification provided here. If you want to do things in a different way than outlined here, talk to a teacher beforehand.

Possible grades for this assignment are U and G (no VG).

Advice and Notes

Free-list

The free-list is a singly linked-list that contains all currently unused elements. A *head*-node is stored separately, referring to the first free element. The last node of the list is assigned a termination value of some kind (see below).
Since insertions (when an object is destroyed) can be made at the *head*, it is sufficient for the list to be singly linked rather than double linked.

In the current design, free-list nodes are stored *inside* the elements of the pool itself, thus requiring no extra memory. This does require however that the element size (`ElementSize`) is **large enough to fit a free-list node**.

Free-list nodes refer to elements. They can do so either via *raw pointers* or as *indices*. Which one to choose is purely a design choice: raw pointers require at least 8 bytes, setting this as a lower limit for `ElementSize`. Indices may be smaller, e.g. `uint16_t`, which allows for a capacity of 2^{16}–1 = 65.535 elements, or 2^{16}–2 if the maximum value is reserved as a null value (for the last free node). The simplest way to obtain the maximum value for an unsigned integer is to assign it to –1, e.g.

```
uint16_t index_null = (uint16_t)-1;
```

Indices have another potential benefit: they are *relative* to the pool's base pointer. If users of the pool maintain *indices* rather than pointers to objects, there is nothing that prevents the pool from expanding by reallocating its main chunk. Since indices are relative, they will not be invalidated by such an event.

Free-list Visitor

For debug purposes, it can be useful to iterate the elements contained by the free-list – and to an extent the *used* elements as well, although we can't know their current types. This can be achieved by a visitor function that iterates the free-list and calls a custom function for each free element. The custom function may for example count free elements, check if a given element exists (is free), or print out its value to `cout`. Below is an example of output produced by a visitor function and different custom function during various pool events:

```
Creating pool with capacity 4...
Pool: [1][2][3][65535]. nbr free 4. freelist head = 0

Creating 2 objects...
Pool: [x][x][3][65535]. nbr free 2. freelist head = 2

Creating 2 objects...
Pool: [x][x][x][x]. nbr free 0. freelist head = 65535

Destroying 2 objects...
~IHaveADestructor
Pool: [x][x][3][65535]. nbr free 2. freelist head = 2

Destroying 2 objects...
Pool: [2][0][3][65535]. nbr free 4. freelist head = 1
```

As might be expected, brackets with a "x" stands for a used element, while brackets with an integer value stands for a free element and the next free element in the list. The test program that produces this output is provided with the handout.

References

[1] Ferres, *Memory management in C: The heap and the stack.* https://cs.gmu.edu/~zduric/cs262/Slides/teoX.pdf (230227)

[2] Gregory, *Game Engine Architecture*, 3rd Edition.