CS 3370

Programming Assignment 2

"Class-based Heaps"

As you know, every direct heap allocation via the `new` operator incurs some overhead so that the `delete` operator will have the information it needs to properly clean up when you're finished with the heap object. When many heap objects are active at once, the wasted memory for the heap overhead for each object can be excessive. One solution is to manage heaps on a custom, class-by-class basis.

The idea is to reserve enough space for many objects with a single heap allocation (a memory "block" capable of holding some number of objects), and then return pointers to locations inside that array yourself as the user requests a new object. You repeat this process as each block of objects gets used up, so you may have many memory blocks active at once. So, you are essentially doing your own heap management for that class. This is library-developer skill often expected of C++ programmers.

In this assignment, you will apply good software design techniques by separating the management of a memory pool from the class of objects to be allocated from the pool. The class of objects here is named `MyObject`, and has the following data members:

```
int id;
string name;
```

To force users to only place `MyObject` objects on the heap, so that you can manage *all* `MyObject` instances, make the constructor **private**, and provide a static member function, `create`, which returns a pointer to a new heap object (this is the Simple Factory design pattern). In other words, the `MyObject` class includes the following member function definitions:

```
private:
   MyObject(int i, const string& nm): name(nm) {            // Note initializer list
      id = i;
   }

public:
   static MyObject* create(int id, const string& name) {    // Factory method
      return new MyObject(id, name);
   }
```

Define also a class named `Pool`, with at least the following member functions:

```
public:
   Pool(size_t blockSize = 5, bool trace_flag = true);
   ~Pool();
   void* allocate();       // Get a pointer inside a pre-allocated block for a new object
   void deallocate(void*); // Free an object's slot (push the address on the "free list")
   void profile() const;   // prints the addresses currently on the free list
```

Provide class-based versions of `operator new` and `operator delete` in `MyObject` that utilize a single, shared (i.e., static) instance of class `Pool`. `Pool` manages a dynamic array of blocks, as discussed above, where each block contains the number of bytes needed to hold `blockSize` elements of type `MyObject`, each of size `elemSize (= sizeof (MyObject)`).

A Pool should be generic; you can have a Pool for any class. The most expedient way to do this it to make it a template. Write the implementation of the Pool functions in the Pool.h file. Note: you may find it desirable to write additional helper functions – they will be private, of course.

The Pool class is the heart of managing the objects' memory (and is the heart of this program.) When the program wants a MyObject object, the request for memory goes to the Pool object associated with MyObject. The Pool object has a bunch of memory that it has got from the Heap; enough for several MyObject instances. When it gets a request for a new MyObject, it returns a pointer to memory for a MyObject. On the other hand, when a MyObject is deleted, the Pool takes the pointer to that memory and can use it later for another allocation. These actions happen inside the allocate() and deallocate() functions mentioned above.

How does the Pool object know which pointer to use for an allocate, and what to do with a deallocated pointer? It keeps a list of the available cell of memory (where each cell is the size of MyObject.) This, of course, is usually called the free list. (Think about it: it doesn't have to keep track of the used chunks of memory, just the free list.) The way to maintain the free list is to use a linked list. For efficiency's sake, when we allocate, we take the first one – the head of the free list. And when we deallocate, it is easiest and fastest to put the freed element on the head of the free list.

When we implement a linked list, we need pointers to the next object in the list. Since we don't want to waste memory, we embed this pointer inside each cell of memory. We can do that because the memory isn't currently being used for anything else. We can force the next pointer into a memory cell using a reinterpret_cast, but there is a way that is easier to understand and leads to code that is also easier to understand.

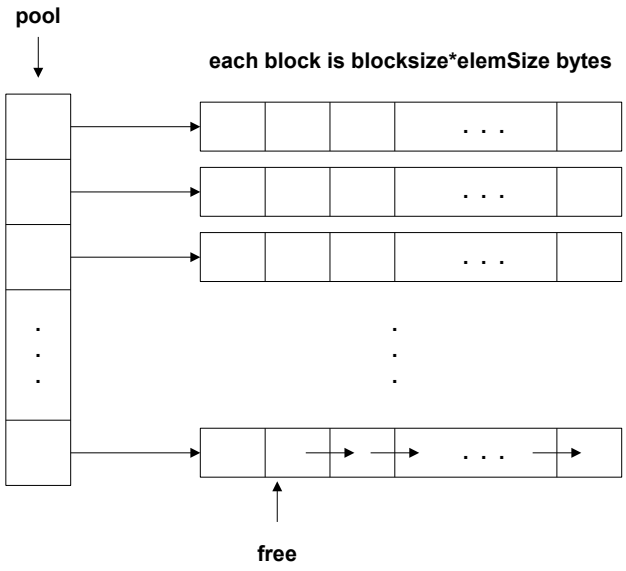
A union is a memory-saving construct that comes all the way from C. It is like a struct, with multiple data members that you declare, but the big difference is that all data members share the same memory! You use the data members just like you do with structs and classes (the dot-operator to access a member). This is exactly what we need. Of course, when you use a union, you have to be careful: if you assign to one of the data members, and then you access a different data member (of a different type, of course), you can get strange results. But that's not an issue in our case.

Create a union called element. It will have two data members. One is a pointer to the next free element. The other data member is just an array of bytes the size of MyObject (or whatever class the Pool is for). The byte array is a placeholder, to make sure that we have just the right amount of memory for one MyObject.

As noted above, the allocate() function returns a pointer to the next available cell of memory (where "cell" is a chunk of memory big enough for one MyObject). And it adjusts its free list. But what if you have run out of available cells? Then the Pool has to go the heap to get more memory, by calling the global operator new(). Of course, it would be silly to get memory for just one MyObject, so we get a block of memory; enough for blockSize MyObjects (see above.) We will get a pointer to contiguous memory. We have to reconstruct our free list with this memory. You can use pointer arithmetic to go through this memory and set up the free list. Remember that the very last element's next pointer must be set to nullptr.

Over time, the Pool might need to get numerous blocks of memory. We will want to clean them all up when the program exits – it's just good citizenship. Therefore, we need to keep track of all these blocks. To do so, we need an array of pointers. Then when the program ends (In the Pool's destructor), we can walk through this array and delete each pointer. But of course, this array of pointers must be dynamic. Every time we get a new block, we need to allocate a slightly larger array of pointers and copy the data, etc., from the old one. You keep track of this array of pointers with a pointer to its start; that would be a pointer to a pointer.

The following is a complete picture of the data:



You will build this data structure implicitly, one row ("block") at a time, as needed. It starts out empty.

The `MyObject` class has a *static* `Pool` data member. Use the inline keyword to cause space for the Pool to be allocated in the class declaration. When `MyObject::operator new` executes, it merely calls `Pool::allocate` (via its static `Pool` object), which in turn returns the pointer to the current open slot in a pool block, and also updates its internal free list pointer to point to the next open slot for the next future call to `allocate`. When there are no more open slots, a `Pool` object automatically expands it array of blocks by 1 more block (or *row*, if you will, as depicted above), and initializes those pointers in logical linked-list fashion as you did in the first instance.

When a `MyObject` is deleted, `MyObject::operator delete` simply calls `Pool::deallocate`, which in turn logically prepends the returned address to the head of the free list that the pool manages (don't shrink the memory allocation in the pool). `Pool`'s destructor needs to delete all heap resources it owns: each block/row, and the array holding the pointers to the blocks. Using **=delete** in the appropriate places, disallow copy and assignment in both `MyObject` and `Pool`. `MyObject` also needs an output operator (<<). See the driver in the file `main.cpp` for an example of how to use these classes. Place the class definitions and member functions for these classes in separate header and implementation files (i.e., `Pool.h, Pool.cpp, MyObject.h, MyObject.cpp`).

This is not a long program (about 150 lines or so). The tedious (but not overly difficult) part is doing the pointer arithmetic and using the union to set up and use the free list. Each block/row is just an array of bytes, so each element in the dynamic array `pool` is a `std::byte*`; therefore, `pool` itself is a `byte**`. You can't allocate an array of `MyObject`s directly for each row in `Pool` because, 1) `Pool` is generic and shouldn't know about `MyObject`, and 2) the default constructor for `MyObject` would execute for each element in each row, which is wasteful – we just want raw, uninitialized memory for us to set up ourselves. The constructor for each `MyObject` runs automatically when the object itself is created when `MyObject::create` invokes the `new` operator, as you know.

Note that you need to insert trace statements in the correct places in `Pool`'s member functions, so your output will be similar to the sample output. Use the default of 5 for `blockSize`. (In practice, of course, it will be much bigger). You will also need a static `MyObject::profile` function that calls a non-static `Pool::profile` function that prints out the free list (see below).

Programming instructions:

- Write everything in pool.h. After all, it's a template.
- I have given you suggestions for private data and a couple of private functions. You may change those if you want. For example, you might not even use the expand() and link() functions.

Here is the expected output (addresses will vary, of course):

```
Initializing a pool with element size 32 and block size 5

Live Cells: 0, Free Cells: 0
Free list:

Expanding pool...
Linking cells starting at 0x7faea0402830
Cell allocated at 0x7faea0402830
Cell allocated at 0x7faea0402850
Cell allocated at 0x7faea0402870
Cell allocated at 0x7faea0402890
Cell allocated at 0x7faea04028b0
Expanding pool...
Linking cells starting at 0x7faea0402920
Cell allocated at 0x7faea0402920
Cell allocated at 0x7faea0402940
Cell allocated at 0x7faea0402960
Cell allocated at 0x7faea0402980
Cell allocated at 0x7faea04029a0
Object 5 == {5,"5"}
Cell deallocated at 0x7faea0402920

Live Cells: 9, Free Cells: 1
Free list:
0x7faea0402920

Creating another object:
Cell allocated at 0x7faea0402920
anotherObject == {100,anotherObject}
Creating yet another object:
Expanding pool...
Linking cells starting at 0x7faea04029c0
Cell allocated at 0x7faea04029c0
yetAnotherObject == {120,yetAnotherObject}

Live Cells: 11, Free Cells: 4
Free list:
0x7faea04029e0
0x7faea0402a00
0x7faea0402a20
0x7faea0402a40

Cell deallocated at 0x7faea0402920
Cell deallocated at 0x7faea04029c0
Cell deallocated at 0x7faea0402830
Cell deallocated at 0x7faea0402850
Cell deallocated at 0x7faea0402870
Cell deallocated at 0x7faea0402890
Cell deallocated at 0x7faea04028b0
Cell deallocated at 0x7faea0402940
Cell deallocated at 0x7faea0402960
Cell deallocated at 0x7faea0402980
Cell deallocated at 0x7faea04029a0

Live Cells: 0, Free Cells: 15
Free list:
0x7faea04029a0
0x7faea0402980
0x7faea0402960
0x7faea0402940
0x7faea04028b0
0x7faea0402890
0x7faea0402870
0x7faea0402850
```

0x7faea0402830
0x7faea04029c0
0x7faea0402920
0x7faea04029e0
0x7faea0402a00
0x7faea0402a20
0x7faea0402a40

Deleting 3 blocks

Assessment Rubric

Competency ↓	**Basic →**	**Proficient →**	**Exemplary**
Memory Management	Every **new** has a **delete** (no memory leaks); destructor does the right thing. Use **delete [ ]** when deleting heap arrays. **operator new** and **operator delete** are declared as static member functions in **MyObject**.	Use **reinterpret_cast** for peeking. **operator new** is private. Alternative: use the union.	Use placement **new** for poking.
Memory efficiency		The only heap memory is the array of pointers to the blocks (called "pool" in the spec), and the blocks themselves. There are no unused elements in the "pool" array of pointers. Blocks are only allocated when absolutely necessary. *Existing blocks are never reallocated after they are created*.	
Clean Code		No repeated code (refactor); no unnecessary code.	Simplest possible logic to fulfill program requirements; only the destructor, profile, and the grow process need a loop.
Defensive Programming			Use **assert** in appropriate places in Pool for design invariants (in the constructor, **allocate**, and **deallocate**)
Other	Use **size_t** for non-negative quantities; static data members are defined in the appropriate .cpp file at file scope. Use **nullptr** instead of 0 (never use NULL!).	Always #include <cstddef> for **size_t**; use #ifndef–#endif guards instead of #pragma once (portability). **operator<<** is a friend to **MyObject**. Use constructor initializer lists for non-built-in types of class members. Use **=delete** as requested.	Use std:: (either as a prefix or in a using declaration) in your header files for names imported from the standard library (*never* use **using namespace std** in header files)