Overloading `new` and `delete`

- Why?
  - Manage memory manually.
    - Tailor allocation and deallocation to provide the best possible performance (time and space).

Overloading `new` and `delete`

- The three most common reasons
  - To detect usage errors
    - Failure to `delete` memory allocated by `new` can result in memory leaks.
      - If `operator new()` keeps a list of allocated addresses and `operator delete()` removes addresses from the list, result is that it is easy to detect this usage error.
    - Programming errors that lead to overruns or underruns.
      - `operator new()` over allocates blocks to store signatures before and after the memory made available to clients. `operator delete()` checks to ensure that the signatures are still intact.

Overloading `new` and `delete`

- The three most common reasons
  - To improve efficiency
    - Versions of `new` and `delete` provided by the standard compiler are generic and work reasonably well for most people and optimally for no one.
    - Custom versions of `new` and `delete` can outperform the default versions computationally and in required storage.
      - Requires a **good** understanding of the program's dynamic memory usage patterns.

Overloading `new` and `delete`

- The three most common reasons
  - To collect usage statistics
    - How does the software use dynamic memory?
      - What is the distribution of allocated block sizes?
      - What is the distribution of their lifetimes?
      - Do they tend to be allocated and deallocated in LIFO, FIFO, or random order?
      - Etc….
      - Custom versions of operator `new` and `delete` facilitate collecting such information.

Overloading `new` and `delete`

- Additional reasons to overload
  - To increase the speed of allocation and deallocation
    - General purpose allocators are often significantly slower than custom versions, especially versions that are tailored to the type of particular objects.
  - To reduce the space overhead of default memory management
    - Not only are general-purpose memory managers often slower, they often use more memory.

Overloading `new` and `delete`

- Additional reasons to overload
  - To compensate for suboptimal alignment in the default allocator.
    - Alignment:
      - Many computer architectures require data of particular types be placed in memory at particular types of addresses. `doubles` may be required to be at addresses that are a multiple of eight (eight-byte aligned)
      - Some compilers do not guarantee eight-byte alignment for `doubles`, which will slow down the program.

Overloading `new` and `delete`

- Additional Reasons to overload
  - To cluster related objects near one another.
    - If particular data structures are generally used together, then can minimize the frequency of page faults when working on the data by creating a separate heap for the data structures.
      - Attempt to minimize the pages needed to store the data.
  - To obtain unconventional behavior.
    - Customize `operator new()` and `operator delete()` to do something that the generic versions do not.
      - For example, `delete` overwrites deallocated memory with zeros to increase the security of the application data.

Overloading `new` and `delete`

- While there are many reasons to overload `operator new()` and `operator delete()`, you should only do it when absolutely necessary!
  - or when the assignment tells you to specifically do it ☺

Overloading Conventions

- Specific conventions must be followed when overloading `new` and `delete`.
- Generally, `new` must have the correct return value, call the new-handling function when insufficient memory is available, and cope with requests for no memory.

Overloading Global `new`: Conventions

- Return the correct value
  - If the requested memory can be provided, then return a pointer to it.
  - If the requested memory cannot be provided, then call the new-handler and throw a `bad_alloc` exception if the memory cannot be allocated.
    - We will not go into the details of new-handler.

Overloading Global `new`: Conventions

- Cope with requests for no memory
  - C++ requires `operator new()` return a legitimate pointer, even when zero bytes are requested.
- Infinite loop
  - The operator should contain an infinite loop that is broken out of when either the memory has been successfully allocated or `bad_alloc` has been thrown.

Overloading Global `new`: Conventions

- Pseudo code

```
void* operator new(std::size_t size) throw(bad_alloc) {
   // Your operator new may take additional params
   using namespace std;
   // handle 0-byte requests by treating them as 1-byte requests
   if (size == 0)
      size = 1;
   while (true) {
      attempt to allocate size bytes;
      if (the allocation was successful)
        return (a pointer to the memory);
      //allocation was unsuccessful; find out what the current
      //new-handling function is
      new_handler globalHandler = set_new_handler(0);
      set_new_handler(globalHandler);

      if (globalHandler)
        (*globalHandler)();
      else
        throw bad_alloc();
   }
}
```

Overloading Member `new`: Conventions

- `operator new()` member functions are inherited by derived classes.
  - Thus, it is possible that the `operator new()` in the base class will be called to allocate memory for an object in a derived class.
    - When this happens, the program needs to call global `operator new()`.

Overloading Member `new`: Conventions

- Pseudo code

```
class Base {
public:
    static void* operator new (std::size_t size) throw(std::bad_alloc);
    …
}
void* Base::operator new (std::size_t size) throw(std::bad_alloc) {
    // if size is incorrect call global operator new, otherwise handle the request
    // here.
    if (size != sizeof(Base))
       return ::operator new(size);
    …
}

class Derived: public Base {
    // Derived doesn't declare operator new
    …
};

Derived *p = new Derived; // calls Base::operator new
```

Overloading `delete`: Conventions

- This is a bit simpler!
  - C++ guarantees that it is always safe to delete the null pointer, so your `delete` function must honor that guarantee.

```
void operator delete(void *rawMemory) throw() {
   // Do nothing if the null pointer is being deleted.
   if (rawMemory == 0)
       return;
   // deallocate the memory pointed to by rawMemory;
}
```

Overloading `delete`: Conventions

- Warning for member `operator delete()` overloading…
  - The `size_t` value passed to `operator delete()` may be incorrect if the object being deleted was derived from a base class lacking a virtual destructor.
    - Need to ensure that base classes have a virtual destructor, or `operator delete()` many not work properly.

Overloading `delete`: Conventions

- Member functions are also simple, but must check the size of what's being deleted.

```
class Base {
public:
    static void* operator new(std::size_t size) throw(std::bad_alloc);
    static void operator delete(void *rawMemory, std::size_t size) throw();
    …
};
void Base::operator delete(void *rawMemory, std::size_t size) throw() {
    if (rawMemory == 0) // check for null pointer
       return;
    // if size is wrong, use the global operator delete to handle the request
    if (size != sizeof(Base)) {
       ::opreator delete(rawMemory);
       return;
    }
    // deallocate the memory pointed to by rawMemory;
    return;
}
```

Memory Pool

- Memory allocation can take a lot of time and memory will fragment over time.
- A Memory Pool allocates a large amount of memory on startup, and will separate this block into smaller chunks. Every time memory is requested from the pool, it is taken from the previously allocated chunks, and not from the OS.
  - The biggest advantages are:
    - Very little (to none) heap-fragmentation
    - Faster than the "normal" memory (de-)allocation (e.g., via `malloc`, `new`, etc.)
  - Additional benefits:
    - Check if an arbitrary pointer is inside the Memory Pool
    - Write a "Heap-Dump" to your HDD (great for post-mortem debugging, etc.)
    - Some kind of "memory-leak detection": the Memory Pool will throw an *`assertion`* when you have not freed all previously allocated memory

Memory Pool

- Steps
  - Pre-allocate Memory
  - Segment the allocated memory
  - Request memory from the memory pool
  - Example
    - Memory_pool.cpp
    - PooledVector.h, PooledVector.cpp
    - UnpooledVector.h, UnpooledVector.cpp