**Chepter-8: Customizing new and delete**

Operator new and operator delete apply only to allocations for single objects. Memory for arrays is allocated by operator new[] and deallocated by operator delete[].

Heap memory for STL containers is managed by the containers’ allocator objects, not by new and delete directly.

**Item 49: Understand the behavior of the new-handler.**

When operator new can’t satisfy a memory allocation request, it throws an exception. Long ago, (old compiler) it returned a null pointer. Before operator new throws an exception in response to an unsatisfiable request for memory, it calls a client-specifiable error-handling function called a new-handler.

To specify the out-of-memory-handling function, clients call set\_new\_handler, a standard library function declared in <new>:

namespace std {

typedef void (\*new\_handler)();

new\_handler set\_new\_handler(new\_handler p) throw();

}

Defined as above, new\_handler is a typedef for a pointer to a function that takes and returns nothing, and set\_new\_handler is a function that takes and returns a new\_handler. (The throw() at the end of set\_new\_handler’s declaration is an exception specification. It essentially says that this function won’t throw any exceptions).

set\_new\_handler’s parameter is a pointer to the function, which is called by operator new if it can’t allocate the requested memory. The return value of set\_new\_handler is a pointer to the function in effect for that purpose before set\_new\_handler was called.

void outOfMem(){ // function to call if operator new can’t allocate enough memory

std::cerr << "Unable to satisfy request for memory\n";

std::abort();

}

int main(){

std::set\_new\_handler(outOfMem);

int \*pBigDataArray = new int[100000000L];

...

}

If operator new is unable to allocate space for 100,000,000 integers, outOfMem will be called, and the program will abort after issuing an error message.

Note: When operator new is unable to fulfill a memory request, it calls the new-handler function repeatedly until it *can* find enough memory (will discuss later).

The newhandler function must do one of the following:

■ **Make more memory available**: This may allow the next memory allocation attempt inside operator new to succeed.

One way to implement this strategy is to allocate a large block of memory at program start-up, then release it for use in the program the first time the new-handler is invoked.

■ **Install a different new-handler**: If the current new-handler can’t make any more memory available, perhaps it knows of a different new-handler that can. If so, the current new-handler can install the other new-handler in its place (by calling set\_new\_handler). The next time operator new calls the new-handler function, it will get the one most recently installed.

(A variation on this theme is for a new-handler to modify its *own* behavior, so the next time it’s invoked, it does something different. One way to achieve this is to have the new-handler modify static, namespace-specific, or global data that affects the new-handler’s behavior.)

■ **Deinstall the new-handler**: Pass the null pointer to set\_new\_handler. With no new-handler installed, operator new will throw an exception when memory allocation is unsuccessful.

■ **Throw an exception:** of type bad\_alloc or some type derived from bad\_alloc. Such exceptions will not be caught by operator new, so they will propagate to the site originating the request for memory.

■ **Not return**, typically by calling abort or exit.

Either of any one choice give above, we can opt while implementing newhandler function.

Sometimes we like to handle memory allocation failures in different ways, depending on the class of the object being allocated:

class X {

public:

static void outOfMemory();

...

};

class Y {

public:

static void outOfMemory();

...

};

X\* p1 = new X; // if allocation is unsuccessful, call X::outOfMemory

Y\* p2 = new Y; // if allocation is unsuccessful, call Y::outOfMemory

C++ has no support for class-specific new-handlers. But we can achieve this via below steps:

* We must provide own versions of set\_new\_handler and operator new for ours class.
* The class’s set\_new\_handler allows clients to specify the new-handler for the class (exactly like the standard set\_new\_handler allows clients to specify the global new-handler).
* The class’s operator new ensures that the class-specific new-handler is used in place of the global new-handler when memory for class objects is allocated.

For Example: If we want to handle memory allocation failures for the Widget class. Means need to keep track of the function to call when operator new can’t allocate enough memory for a Widget object. For that need to declare a static member of type new\_handler to point to the new-handler function for the class. Widget will look something like this:

class Widget {

public:

static std::new\_handler set\_new\_handler(std::new\_handler p) throw();

static void\* operator new(std::size\_t size) throw(std::bad\_alloc);

private:

static std::new\_handler currentHandler;

};

Static class members must be defined outside the class definition (unless they’re const)

std::new\_handler Widget::currentHandler = 0;

The set\_new\_handler function in Widget will save whatever pointer is passed to it, and it will return whatever pointer had been saved prior to the call. This is what the standard version of set\_new\_handler does:

std::new\_handler Widget::set\_new\_handler(std::new\_handler p) throw()

{

std::new\_handler oldHandler = currentHandler;

currentHandler = p;

return oldHandler;

}

Finally, Widget’s operator new will do the following:

1. Call the standard set\_new\_handler with Widget’s error-handling function. This installs Widget’s new-handler as the global newhandler.
2. Call the global operator new to perform the actual memory allocation. If allocation fails,

the global operator new invokes Widget’s new-handler, because that function was just installed as the global new-handler. If the global operator new is ultimately unable to allocate the memory, it throws a bad\_alloc exception. In that case, Widget’s operator new must restore the original global newhandler, then propagate the exception.

1. If the global operator new was able to allocate enough memory for a Widget object, Widget’s operator new returns a pointer to the allocated memory. The destructor for the object managing the global new-handler automatically restores the global new-handler

to what it was prior to the call to Widget’s operator new.

Now must design resource-handling class, which consists of nothing more than the fundamental RAII operations of acquiring a resource during construction and releasing it during destruction.

class NewHandlerHolder {

public:

// acquire current new-handler

explicit NewHandlerHolder(std::new\_handler nh) : handler(nh) {}

~NewHandlerHolder(){ // release it

std::set\_new\_handler(handler);

}

private:

std::new\_handler handler; // remember it

NewHandlerHolder(const NewHandlerHolder&); // prevent copying

NewHandlerHolder& operator=(const NewHandlerHolder&);

};

This makes implementation of Widget’s operator new quite simple:

void\* Widget::operator new(std::size\_t size) throw(std::bad\_alloc)

{

// install Widget’s new-handler

NewHandlerHolder h(std::set\_new\_handler(currentHandler)); return ::operator new(size); // allocate memory or throw

} // restore global new-handler.

Clients of Widget use its new-handling capabilities like this:

void outOfMem(); // decl. of func. to call if mem. alloc. for

// Widget objects fails

Widget::set\_new\_handler(outOfMem); // set outOfMem as Widget’s new-handling

// function

Widget \*pw1 = new Widget; // if memory allocation fails, call outOfMem

std::string \*ps = new std::string; // if memory allocation fails, call the global new-

// handling function (if there is one)

Widget::set\_new\_handler(0); // set the Widget-specific new-handling function

// to nothing (i.e., null)

Widget \*pw2 = new Widget; // if mem. alloc. fails, throw an exception

// immediately. (There is no new- handling

// function for class Widget.

**Things to Remember**

✦ set\_new\_handler allows you to specify a function to be called when memory allocation

requests cannot be satisfied.

✦ Nothrow new is of limited utility, because it applies only to memory allocation;

associated constructor calls may still throw exceptions.

**Item 50: Understand when it makes sense to replace new and delete.**

Why we need replace the compiler-provided versions of operator new or operator delete?

These are three of the most common reasons:

■ **To detect usage errors:** Using more than one delete on newed memory leads undefined behavior. If operator new keeps a list of allocated addresses and operator delete removes addresses from the list, it’s easy to detect such usage errors.

Problem due to data overruns (writing beyond the end of an allocated block) and underruns (writing prior to the beginning of an allocated block). Custom operator news can

Over allocate blocks so there’s room to put known byte patterns (“signatures”) before and after the memory made available to clients.

■ **To improve efficiency:** Default Operator new and delete bundled with compiler for general purpose uses:

* They must be acceptable for long-running programs, also for programs that execute for less than a second.
* They must handle series of requests for large blocks of memory, small blocks, and mixtures of both.
* They are worry about heap fragmentation (a process that unable to satisfy requests for large blocks of memory, even when ample free memory is distributed across many small blocks).
* And many others.

All those given above demands are fulfill by operator new and delete which comes by default with compiler. They work reasonably well for everybody, but optimally for nobody because they follow middle-of-the-road strategy.

If we have a good understanding of our program’s dynamic memory usage patterns then, we

can customize operator new and delete for better performance. Better performance means they run faster, they require less memory — up to 50% less.

For some applications, replacing the stock new and delete with custom versions is an easy way to pick up significant performance improvements.

■ **To collect usage statistics:**

Custom versions of operator new and operator delete make it easy to collect these kinds of information:

* What is the distribution of allocated block sizes?
* What is the distribution of their lifetimes?
* Do they tend to be allocated and deallocated in FIFO order, LIFO or something closer to random order?
* Do the usage patterns change over time, e.g., does your software have different allocation/deallocation patterns in different stages of execution?
* What is the maximum amount of dynamically allocated memory in use at any one time?

For example, writing an operator new that facilitates the detection of under- and overruns.

static const int signature = 0xDEADBEEF;

typedef unsigned char Byte;

// this code has several flaws — see below

void\* operator new(std::size\_t size) throw(std::bad\_alloc)

{

using namespace std;

size\_t realSize = size + 2 \* sizeof(int); // increase size of request so 2

// signatures will also fit inside

void \*pMem = malloc(realSize); // call malloc to get the actual memory

if (!pMem) throw bad\_alloc();

// write signature into first and last parts of the memory

\*(static\_cast<int\*>(pMem)) = signature;

\*(reinterpret\_cast<int\*>(static\_cast<Byte\*>(pMem)+realSize-sizeof(int))) =signature;

// return a pointer to the memory just past the first signature

return static\_cast<Byte\*>(pMem) + sizeof(int);

}