**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);

}

One Problem is: Operator news should contain a loop calling a new-handling function, but this one doesn’t. *We will cover this in next item.*

*The alignment problem:* Different computer has different architectures. For example, A 32-bit machine might require that pointers occur at addresses that are a multiple of four (i.e., be four-byte aligned) or 64-bit machine required addresses which is multiple of eight (i.e., be eight-byte aligned). Failure to follow such constraints could lead to hardware exceptions at runtime or some performance bottleneck.

Alignment is relevant here, because C++ requires that all operator news return pointers that are suitably aligned for *any* data type.

Operator new return a pointer it gets from malloc is safe. However, in operator new above, we’re not returning a pointer we got from malloc, we’re returning a pointer we got from malloc *offset by the size of an* int.

There is no guarantee that this is safe! If the client called operator new to get enough memory for a double and we were running on a machine where ints were four bytes in size but doubles were required to be eight-byte aligned, we’d probably return a pointer with improper alignment. That might cause the program to crash. Or it might just cause it to run more slowly. Either way, it’s probably not what we had in mind.

*Writing a custom memory manager is almost easy but writing one that works efficiently is a lot harder. Generally, suggested not to attempt it unless we have to.*

**Things to Remember**

✦ There are many valid reasons for writing custom versions of new and delete, including improving performance, debugging heap usage errors, and collecting heap usage information.

**Item 51: Adhere to convention when writing new and delete.**

In this Item we will discuss the rules must be follow when we write custom new & delete.

1. Calling the new-handling function when insufficient memory is available.
2. Return the requested memory pointer.
3. If cannot allocated the memory, install different handler and again try (Item 49). Still not then throw an exception of type bad\_alloc.

Operator new tries to allocate memory more than once, calling the new-handling function after each failure. The assumption here is that the new-handling function might be able to do something to free up some memory. Only when the pointer to the new-handling function is null does operator new throw an exception.

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

{ // your operator new might

using namespace std; // take additional params

if (size == 0) // handle 0-byte requests

size = 1; // by treating them as

// 1-byte requests

while (true) {

attempt to allocate size bytes;

if (the allocation was successful)

return (a pointer to the memory);

if(allocation was unsuccessful)

find out what the current new-handling function is (see below)

new\_handler globalHandler = set\_new\_handler(0);

set\_new\_handler(globalHandler);

if (globalHandler)

(\*globalHandler)();

else

throw std::bad\_alloc();

}

}

Here’s pseudocode for a non-member operator delete C++ guarantees it’s always safe to delete the null pointer. On a global scope do not required the argument size.

void operator delete(void \*rawMemory) throw()

{

if (rawMemory == 0) return; // do nothing if the null

// pointer is being deleted

deallocate the memory pointed to by rawMemory;

}

//Global operator new overload.

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

while (true) {

void \*pMem = malloc(size); //Allocate the memory

if (pMem)

return pMem; //return if success.

//Get a new handler. There is only one way to get & set new

//handler. Set\_new\_handler(0) return the current handler.

std::new\_handler currentHandler = std::set\_new\_handler(0);

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

if (currentHandler)

(\*currentHandler)(); //Invoke new handler

else

throw std::bad\_alloc(); //If new handler is null,

//throw exception.

}

}

Operator new contains an infinite loop, and the code above has infinite loop. The only way out of the loop is for memory to be successfully allocated or throw the exception when unbale to allocate the memory. Make more memory available, install a different new-handler, deinstall the new-handler, throw an exception bad\_alloc, or fail to return. It should now be clear why the new-handler must do one of those things. If it doesn’t, the loop inside operator new will never terminate.

//Class scope operator new overload.

Problem in inheritance.

Because of inheritance, however, it is possible that the operator new in a base class will be called to allocate memory for an object of a derived class:

class Base {

public:

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

...

};

class Derived: public Base // Derived doesn’t declare operator new

{ ...

};

Derived \*p = new Derived; // calls Base::operator new!

How we overcome from this problem? Solution is:

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

{

if (size != sizeof(Base)) // if size is “wrong,”

return ::operator new(size); // have standard operator

// new handle the request

... // otherwise handle

// the request here

}

Similarly, for delete (class scope) would require addition parameter size. The size\_t value C++ passes to operator delete may be incorrect if the object being deleted was derived from a base class lacking a virtual destructor. This is reason enough for making sure base classes have virtual destructors.

void Base::operator delete(void \*rawMemory, std::size\_t size) throw()

{

if (rawMemory == 0) return; // check for null pointer

if (size != sizeof(Base)) { // if size is “wrong,”

::operator delete(rawMemory); // have standard operator

return; // delete handle the request

}

deallocate the memory pointed to by rawMemory;

return;

}

Below code Example:

class Dog {

public:

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

if (size == sizeof(Dog)) //This is why? because of YellowDog

//having diffrent size

costomNewForDog();

else

::operator new(size); //For that we called global

//operator new. this is one

//solution

//Other one is, we can define operator new for YellowDog as well //as.

}

static void operator delete(void \*vptr, std::size\_t size)throw() {

if(vptr==null) return;

if(size!=sizeof(Dog){

::operator delete(vptr);

Return;

}

costomDeleteForDod();

free(vptr);

}

virtual ~Dog() {} //Solution, when base class pointer

//hold derived class object.

};

class YellowDog:public Dog {

int age;

public:

static void operator delete(void \*vptr) {

if(vptr==null) return;

costomDeleteForYellowDod();

free(vptr);

}

};

**Things to Remember**

✦ operator new should contain an infinite loop trying to allocate memory, should call the new-handler if it can’t satisfy a memory request, and should handle requests for zero bytes. Class-specific versions should handle requests for larger blocks than expected.

✦ operator delete should do nothing if passed a pointer that is null. Class-specific versions should handle blocks that are larger than expected.

**Item 52: Write placement delete if you write placement new.**

When we write a new expression like:

Widget \*pw = new Widget;

Two functions are called: one to operator new to allocate memory, a second to Widget’s default constructor. Suppose that the first call succeeds, but the second call results in an exception being thrown. In that case, the memory allocation performed in step 1 must be undone (memory leak).

And there is no way to handled leaked memory by client code because memory address is not caught.

Now, the responsibility for undoing step 1 must therefore fall on the C++ runtime system.

The runtime system is happy to call the operator delete that corresponds to the version of operator new it called in step 1.

Normal signature of operator new:

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

corresponds to the normal operator delete.

void operator delete(void \*rawMemory) throw(); // normal signature at global scope

void operator delete(void \*rawMemory, std::size\_t size) throw();// typical normal

//signature at class scope

We can also declare non-normal forms of operator new — which takes additional parameters. For example, suppose we write a class-specific operator new that requires additional parameter ostream for logging purpose, corresponding required to write a class-specific operator delete(which is not written as of now).

class Widget {

public:

...

// non-normal form of new

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

// normal class specific form of delete

static void operator delete(void \*pMemory, std::size\_t size) throw();

...

};

This design is problematic. (non-normal form of new required equal and opposite behavior of operator delete. Here we have standard operator delete function)

When an operator new function takes extra parameters (other than the mandatory size\_t argument), that function is known as a *placement* version of new.

A particularly useful placement new is the one that takes a pointer specifying where an object should be constructed. That operator new looks like this:

void\* operator new(std::size\_t, void \*pMemory) throw(); // “placement new”

It is an original placement new syntax which is available in #include<new>.

Many of us gives the placement new definition “It is a specific function, operator new taking a single extra argument of type void\*.” But it is not completely true. In a general term “placement new” means any version of new taking extra arguments (phrase “placement delete” also derived from here).

Now, get back and the problem statement is “Above class design is problematic” but Why?

Consider this client code, which logs allocation information to cerr when dynamically creating a Widget:

Widget \*pw = new (std::cerr) Widget; // call operator new, passing cerr as the

// ostream; this leaks memory if the Widget

// constructor throws

Suppose, if memory allocation succeeds and the Widget constructor throws an exception, the runtime system is responsible for undoing the allocation that operator new performed. The runtime system can’t really understand how the called version of operator new works, so it can’t undo the allocation itself. Instead, the runtime system looks for a version of operator delete that takes the same number and types of extra arguments as operator new, and, if it finds it, that’s

the one it calls.

Here the corresponding operator delete signature is:

void operator delete(void\*, std::ostream&) throw();

The versions of operator delete that take extra parameters are known as placement deletes.

In this case, Widget declares no placement version of operator delete, so the runtime system doesn’t know how to undo what the call to placement new does. As a result, it does nothing. In this example, *no* operator delete *is called* if the Widget constructor throws an exception!

To eliminate the memory leak in the code above, Widget needs to declare a placement delete that corresponds to the logging placement new:

class Widget {

public:

...

static void\* operator new(std::size\_t size, std::ostream& logStream)

throw(std::bad\_alloc);

static void operator delete(void \*pMemory) throw();

static void operator delete(void \*pMemory, std::ostream& logStream) throw();

...

};

With this change, if an exception is thrown from the Widget constructor in this statement,

Widget \*pw = new (std::cerr) Widget; // as before, but no leak this time

The corresponding placement delete is automatically invoked, and that allows Widget to ensure that no memory is leaked. However, consider what happens if no exception is thrown (which will usually be the case) and we get to a delete in client code:

delete pw; // invokes the normal operator delete not placement delete.

This means to prevent all type of memory leaks associated with placement versions of new, we must provide both the normal operator delete (for when no exception is thrown during construction) and a placement version that takes the same extra arguments as operator new does.

**Things to Remember**

✦ When you write a placement version of operator new, be sure to write the corresponding placement version of operator delete. If you don’t, your program may experience subtle, intermittent memory leaks.

✦ When you declare placement versions of new and delete, be sure not to unintentionally hide the normal versions of those functions.