Exception Safe Memory Management

Gregory Colvin
Information Management Research
gregor@netcom.com

Traditionally, C++ memory management did not throw exceptions. While the default *operator new* functions could not allocate memory they called the function installed by *set_new_handler()* if any, or else returned 0. By default there was no new-handler.

In a break with existing practice, we have changed the semantics of the default *operator new* functions by specifying a default new-handler which throws an *alloc* exception. If no new-handler is installed (that is if *set_new_handler(0)* is called) the results are undefined, although we note that this might reasonably restore the traditional behavior. This change breaks existing code, leaves undefined what used to be a well defined behavior, and makes it difficult to write new code which is exception safe. Where we used to write

```
void *p= new T;
if (p == 0)
    handle_out_of_memory();
```

we must now, in defense against both a possibly missing new-handler and a possible exception, write

```
try {
   void *p= new T;
   if (p == 0)
      handle_out_of_memory();
} catch (alloc) {
   handle_out_of_memory();
}
```

To further complicate matters, the relationship of the STL *allocator* template to the *operator new* functions, and whether STL *allocator*s are required to throw exceptions, is currently unspecified.

The following proposal is based liberally on ideas taken from Fergus Henderson, Nathan Meyers, Richard Minner, John Skaller, and others. I have attempted to specify a memory management facility which provides both the new and the old semantics, can be replaced easily and customized flexibly by advanced users, and can be used simply and safely by any user. Under this proposal the user who simply invokes *new* expecting a possible *alloc* exception need change nothing, the user who wishes to retain the traditional semantics can invoke *new(nothrow())*, and the user who wishes to globally replace the default allocation and deallocation functions can still do so.

Interface

```
void (*set_new_handler(void (*handler)()))();
bool new_handler();

void* allocate(size_t) throw();
void deallocate(void*) throw();

void* operator new(size_t) throw(alloc);
void* operator new[](size_t) throw(alloc);

void operator delete(void*) throw();
void operator delete[](void*) throw();

class nothrow {};
void* operator new(size_t,const nothrow&) throw();
void* operator new[](size_t,const nothrow&) throw();

template<class X> void* operator new(size_t,const X&) throw(X);
template<class X> void* operator new[](size_t,const X&) throw(X);
```

Semantics

The function
```
void (*set_new_handler(void (*handler)()))();
```
sets the current new-handler to *handler*, returning the previous new-handler. By default there is no new-handler, so the call *set_new_handler(0)* restores the default state of the new-handler. The function
```
bool new_handler();
```
calls the current new-handler if any and returns true, or else returns false. It is a convenience for writing allocation functions, which would otherwise have to call *set_new_handler()* once to get the current handler, then call it again to restore the current handler.

The default memory allocation function is
```
void* allocate(size_t) throw();
```
which returns a pointer to the requested memory, if possible. While memory is unavailable it calls the current new-handler if any, or else returns 0, just like the traditional *operator new*.

The default memory invalidation function is
```
void deallocate(void*) throw();
```
which takes as a parameter either 0 or a value returned by *allocate()*.

The default *operator new* functions obtain memory by calling *allocate()*, and the default *operator delete* functions invalidate memory by calling *deallocate()*. The default STL *allocator* also obtains and invalidates memory using *allocate()* and *deallocate()*, and throws *alloc* if memory cannot be obtained. Thus a user can still replace all default memory management by replacing just two functions.

The default functions
```
void* operator new(size_t n) throw(alloc);
void* operator new[](size_t n) throw(alloc);
```
throw an *alloc* exception when *allocate(n)* returns 0. Note that as a transition aid an implementation may define an extension to cause these functions to return 0 to report memory exhaustion, as is traditional.

The default functions
```
void operator delete(void* p) throw();
void operator delete[](void* p) throw();
```
call *deallocate(p)* to invalidate memory obtained from *allocate()* by the corresponding *operator new*.

A empty class is provided to serve as a parameter to two default functions
```
class nothrow {};
void* operator new(size_t n,const nothrow&) throw();
void* operator new[](size_t n,const nothrow&) throw();
```
which return 0 when *allocate(n)* returns 0. Thus calls like *new(nothrow()) T()* and *new(nothrow()) T[n]* can be used to obtain memory without risking an exception throw, as is traditional.

The function templates
```
template<class X> void* operator new(size_t n,const X& x) throw(X);
template<class X> void* operator new[](size_t n,const X& x) throw(X);
```
obtain memory by calling *allocate(n)* and throw the *exception x* if it returns 0. The class *X* must be derived from *exception*. One use for these templates is to throw an exception whose message reports the actual location where *new* was invoked, as opposed to just a location within *new*.

Late Breaking Alternative

If Nathan Meyer's proposal (X3J16/94-0161, WG21/N0548) is accepted then I would propose that class *nothrow* be an *allocator*, and that the *nothrow* placement new functions and the *exception* function templates be subsumed by the *allocator* placement new functions:
```
void* operator new(size_t,allocator&);
void* operator new[](size_t,allocator&);
```