parameter that suppresses throwing an exception on allocation failure), then we can easily emulate it:

```
//in our custom header file "mynew.h"
struct nothrow { }; // dummy struct
void* operator new(size_t, nothrow) throw();
..
..
// implementation in a program
#include "mynew.h"
..
void* operator new(size_t s, nothrow) throw()
{
  void* ptr;
  try {
    ptr = operator new(s);
  }
  ..

  catch(...) { // ensure it does not propagate the exception
   return 0;
  }
return ptr;
}//end new
```

If `ptr = new X` is used in a program, it results in a call of the "normal" operator `new` (the one that throws exceptions on failure), whereas if `ptr = new(nothrow) X` is used, it results in a call to the specialized `new` that is guaranteed to return a pointer and throw no exceptions.

The term "placement syntax" comes from the most intended use of the extra argument(s): to be used to "place" an object to a special location. This particular overload of `new` can be found in the Standard Library. It takes a void pointer to a segment of memory as an additional argument, and is intended to be used whenever memory for the object has already been allocated or reserved by other means. This so-called placement-`new` does not allocate memory; instead, it uses the memory passed to it (the programmer is responsible for ensuring the memory segment is big enough) and then calls the appropriate constructor. We will discuss some of the applications of placement-`new` in the next chapter.

The operator `new` is intended for dynamic creation of individual objects. For creation of arrays of objects, the operator `new[size_t]` must be