Document Number:	X3J16/91-0126, WG21/N0059
Date:	11/29/91
Project:	Programming Language C++
Ref Doc:	X3J16/90-0077
Reply To:	Michael J. Vilot mjv@objects.mv.com

18.1 Language Support

The classes and functions in this section are required to support certain aspects of the C++ language.

18.1.1 Free Store <new>

These functions support the Free Store management described in Section 5.3 and Chapter 12.

18.1.1.1 operator new()

```
void* operator new(size_t) throw(xalloc);
void* operator new(size_t, void*);
```

When an object is created with the new operator, an operator new() function is used (implicitly) to obtain the store needed.
For array allocation, the implementation calculates the storage required to hold the array and calls operator new() with the resulting size.
The second form of operator new() is one of an infinite set of overloaded functions for use with *placement expressions*. The default version provided in the library returns its second argument as its result,

If operator new() is called with the argument 0, it will return a unique address. The results of dereferencing this pointer are ***undefined***.

If operator new() cannot find storage to return, it checks the current *new-handler* [§18.1.1.3]:
If there is a *new-handler*, operator new() will call it and make another attempt to allocate memory.
Otherwise, it will return null.

18.1.1.2 operator delete()

```
void operator delete(void*) throw(xalloc);
```

The **delete** operator destroys an object created by the **new** operator, and (implicitly) calls the operator delete() function to release the storage allocated by operator new().

The operand of **delete** must be a pointer returned by **new**.
The effect of applying **delete** to a pointer not obtained from **new** without a *placement specification*, or applying **delete** to a pointer already deleted, is to throw an xalloc exception [§18.1.2.7].

Deleting a null pointer is harmless.

When **new** allocates an array it must store the number of elements of the array. The information thus stored away is retrieved and used by the **delete** operator.

18.1.1.3 set_new_handler()

```
void (*)() set_new_handler ( void (*)() );
```

The previous function given to `set_new_handler()` will be the return value; this enables users to implement a stack strategy for using *new–handlers*. [§18.1.2.5]

The default *new-handler* function throws an `xalloc` exception [§18.1.2.7].

> *Adopting this section of the proposal requires the following change to RM(5.3.3 / 10): "Any form of operator new() may indicate failure to allocate storage by throwing an exception. If it returns 0 the effect is implementation defined."*

□ Note that the *new-handler* function is anonymous — it cannot be called directly. To obtain the current *new-handler*, call `set_new_handler()` with a known argument (for example, 0), save the result, and call `set_new_handler()` again with the result to re-set the *new-handler* back to what it was.

□ Earlier implementations provided no default *new-handler*. causing *new expressions* to return null when the memory request could be met. C++ programs that used the result of *allocation expressions* without checking the result were erroneous, while those that checked were correct. Providing a default *new-handler* that throws an exception "fixes" the erroneous programs, but breaks the correct ones (in that it requires them to do the checking with a *catch-clause*). The old behavior can be restored by calling `set_new_handler(0)`.

18.1.2 Exceptions <exception>

These functions support the Exception Handling described in Chapter 15.

The functions `terminate()` and `unexpected()` help cope with errors related to the exception handling mechanism itself.

18.1.2.1 terminate()

```
void terminate();
```

This function is called when exception handling must be abandoned. For example,

- when the exception handling mechanism cannot find a handler for a thrown exception,
- when the exception handling mechanism finds the stack corrupted, or
- when a destructor called during stack unwinding caused by an exception tries to exit using an exception.

`terminate()` calls the last function given as an argument to `set_terminate()`.
The default function called by `terminate()` is `abort()` [§18.3.*TBD*].

18.1.2.2 unexpected()

```
void unexpected()
```

If a function with an *exception-specification* throws an exception that is not listed in the *exception-specification*, the function `unexpected()` is called.

`unexpected()` calls the last function given as an argument to `set_unexpected()`.
The default function called by `unexpected()` is `terminate()`.

18.1.2.3 set_terminate()

```
void (*)() set_terminate( void (*)() );
```

The previous function given to `set_terminate()` will be the return value; this enables users to implement a stack strategy for using `terminate()` [§18.1.2.5].

18.1.2.4 set_unexpected()

```
void (*)() set_unexpected( void (*)() );
```

The previous function given to `set_unexpected()` will be the return value; this enables users to implement a stack strategy for using `unexpected()` [§18.1.2.5].

18.1.2.5 stack_handler template

```
template< void (*)() Function( void (*)() ) >
class stack_handler {
public:
  stack_handler( void (*)() );
 ~stack_handler();
private:
  // implementation-defined
};

typedef stack_handler<set_new_handler>    stack_new_handler;
typedef stack_handler<set_terminate>      stack_terminate_handler;
typedef stack_handler<set_unexpected>     stack_unexpected_handler;
```

The constructor calls `Function()` with its argument and stores the result.
The destructor calls `Function()` with the stored result to restore the old value.

18.1.2.6 xmsg exception

```
class xmsg {
public:
  xmsg(string msg);

  string why() const;
  void raise() throw(xmsg);
private:
  // implementation-defined
};
```

The absence of a default constructor means that every `xmsg` must contain a meaningful message.
`x.why()` is the string used to construct an `xmsg x`. `xmsg(s).why() == s`.

□ The intent of the `xmsg` exception class was to allow programs to catch all exceptions in the library:

```
#include <stdlib.h>
#include <iostream>

int main(int argc, char** argv)
{
  try {
    real_main(argc,argv);
    return EXIT_SUCCESS;
  } catch( xmsg& m) {
    cerr << "exiting because of exception: " << m.why() << endl;
    return EXIT_FAILURE;
  }
}
```

`xmsg::raise()` adds no functionality but is included as a convenient hook for debugging.
It is defined by:

```
void xmsg::raise() throw(xmsg) { throw *this; }
```

18.1.2.7 xalloc exception

```
class xalloc : public xmsg {
public:
  xalloc(string msg, size_t size);

  size_t requested() const;
  void raise() throw(xalloc);
private:
  // implementation-defined
};
```

An `xalloc` exception is thrown by the default *new-handler* when `operator new` cannot find storage to allocate [§18.1.1.3].
An `xalloc` exception is thrown when `operator delete` is called with an address not generated from `operator new` [§18.1.1.2].

❑ The standard does not define the form of an `xalloc` error message. The following might be plausible (subject to locale settings):

```
msg + ": Insufficient space to allocate " + int_to_string(size) + " bytes"
```

However, since the `xalloc` exception is going to be thrown when the system runs out of space, the space for constructing and throwing the exception must exist. This implies the error message cannot rely on a string catenation operation that attempts to allocate storage.

A plausible implementation would be an allocator that holds back enough storage, such as a static instance of an `xalloc` object.