Document Number: X3J16/92-0084, WG21/N0161
Date: 9/20/92
Project: Programming Language C++
Ref Doc: X3J16/92-0043, WG21/N0120
Reply To: Michael J. Vilot
`mjv@objects.mv.com`

17. Library

Functions defined in the standard library are *reserved* to the implementation. The behavior of a C++ program that defines functions with signatures matching any of the reserved functions is ***undefined***.

- ❑ Reentrancy: The intent is to allow the library to be reentrant, despite the implied existence of single, global state information (such as `cin`, `cout`, `cerr`, *new-handler*, *unexpected-function*, and *terminate-function*). Multi-threaded implementations will need to provide the appropriate concurrency interlocks if they support a single, global state. Alternatively, they may provide separate copies for each thread.

17.1 Language Support

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

17.1.1 Free Store <new>

These functions support the Free Store management described in Section 5.3 and Chapter 12.
The implementation calls `::operator new()` to support *new-expressions* [§5.3.3], and calls `::operator delete()` to support *delete-expressions* [§5.3.4].

The functions `::operator new(size_t)` and `::operator delete(void*)` are not reserved.
A C++ program may provide at most one definition of each of these functions.

- ❑ Any such functions will replace the default versions. This replacement is global and takes effect upon program startup [§3.4]. Any parts of the implementation (including other portions of the standard library) that rely on `::operator new()` and `::operator delete()` for free store management will use the supplied replacements.

The versions of `::operator new()` and `::operator delete()` described here are the default versions supplied by an implementation.
Any C++ program that replaces `::operator new()` and/or `::operator delete()` with functions having different result semantics causes that program to have ***undefined*** behavior.

The relationship between these memory management functions and the functions `malloc()`, `calloc()`, `realloc()` and `free()` [§17.4.10.3] is ***unspecified***.

17.1.1.1 `operator new()`

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

When an object is created with the **new** operator, the implementation uses `::operator new()` 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.
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.

- ❑ This implies that it is **not** the responsibility of `::operator new()` and `::operator delete()` to store and retrieve this information — they deal strictly in terms of contiguous regions of storage.

Result semantics:

If successful, returns a pointer to allocated storage.
Otherwise, throws an `xalloc` exception [§17.1.2.3.2].
Any other action is ***undefined***.

- ❑ Since the exception will be propagated through a *new-expression*, it changes the semantics of such expressions. Error handling now relies on a catch clause, not a null pointer result.
  Returning a null pointer is undefined, but allowed to ease transition from earlier language implementations.

Adopting this proposal requires the following change to §5.3.3/11:
"Any form of `operator new()` may indicate failure to allocate storage by throwing an `xalloc` exception [§17.1.2.3.2]. If it returns `0`, the effect is implementation defined."

The order and contiguity of storage allocated by successive calls to `operator new()` is ***unspecified***.
The initial value of this storage is ***unspecified***.
The pointer returned is suitably aligned so that it may be assigned to a pointer of any type and then used to access such an object or an array of such objects in the storage allocated (until the storage is explicitly deallocated by a call to `operator delete()` [§17.1.1.2]).
Each such allocation shall yield a pointer to storage disjoint from any other allocated storage.
The pointer returned points to the start (lowest byte address) of the allocated storage.
If the size of the space is requested is 0, the value returned shall be a unique pointer. Repeated such calls return distinct non-null pointers. [§5.3.3]
The results of dereferencing any of these pointers are ***undefined***.

- ❑ The wording for the above 8 sentences was adapted from §17.4.10.3. The intent is to have `operator new()` implementable by calling `malloc()` or `calloc()`, so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

Description of default implementation:

1) Attempts to allocate storage to hold at least the amount of storage requested.

- ❑ Note that the actual size may be larger than the requested size, due to alignment requirements. Some implementations convert a request for 0 bytes into a request for 1 byte.

If successful, returns the address of storage allocated.

2) If unsuccessful, checks the current *new-handler* [§17.1.1.4]:
If there is no *new-handler* installed, throws an `xalloc` exception [§17.1.2.3.2].
Otherwise, calls the *new-handler*.

- ❑ The installed *new-handler* may throw an `xalloc` exception.

3) If the call to the *new-handler* returns, makes another attempt to allocate memory (1 above).

17.1.1.2 **`operator delete()`**

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

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()`.

Result semantics:

Applying `::operator delete()` to a null pointer has no effect.

The argument to `::operator delete()` must be a pointer returned by `::operator new()` [§17.1.1.1].
The effect of applying `::operator delete()` to a pointer not obtained from `::operator new()` is ***undefined***.

The value of a pointer that refers to deallocated space is ***indeterminate***.
The effect of dereferencing a pointer already deleted is ***undefined***.
The effect of applying `::operator delete()` to a pointer already deleted is ***undefined***.

- ❑ An implementation could (should) throw an exception if it can detect these conditions.

Description of default implementation:

Deallocates the storage referenced by the pointer.

17.1.1.3 *placement* operator new()

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

This function is reserved.

- ❑ This second form of `::operator new()` is one of an unbounded set of overloaded functions for use with *placement expressions.*

The placement version of `operator new()` returns its second argument as its result:

```
void* operator new(size_t, void* p) { return p; }
```

17.1.1.4 *new-handler* function

```
typedef void (*new_handler)() throw(xalloc);
```

When `::operator new()` [§17.1.1.1] cannot allocate storage to satisfy a request, it calls the currently installed *new-handler* function.
A C++ program may install *new-handler* functions via calls to `set_new_handler()` [§17.1.1.5].

Result semantics:

A *new-handler* function shall either
1) return after deallocating some currently-allocated storage, or
2) throw an `xalloc` exception or an exception derived from `xalloc` [§17.1.2.3.2], or
3) call `abort()` [§17.4.10.4.1] or `exit()` [§17.4.10.4.3].

Any C++ program that installs a *new-handler* having different result semantics causes that program to have ***undefined*** behavior.

- ❑ In particular, a *new-handler* that returns to the default `::operator new()` [§17.1.1.1] without freeing any storage will cause an infinite loop.

Description of default implementation:

The default *new-handler* function throws an `xalloc` exception [§17.1.2.3.2]:

```
void __new_handler() { throw xalloc; }
```

- ❑ Earlier implementations provided no default *new-handler*, causing *new expressions* to return null when the memory request could not be met. C++ programs that used the result of *new 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 (by detecting all memory exhaustion conditions), but breaks the previously correct ones (by requiring them to do the checking with a *catch-clause*). The old behavior can be restored by calling `set_new_handler(0)`.

17.1.1.5 `set_new_handler()`

```
new_handler set_new_handler ( new_handler );
```

Installs the function given as an argument as the current *new-handler* [§17.1.1.4].

Returns the previous function given to `set_new_handler()`.

❑ This enables callers to implement a stack strategy for using *new–handlers*. 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.

17.1.2 Exceptions `<exception>`

These functions support the Exception Handling described in Chapter 15.

17.1.2.1 `Abnormal termination`

These functions allow C++ programs to control how the implementation responds to faults in the exception handling mechanism.

17.1.2.1.1 `terminate()`

```
void terminate();
```

This function is called when exception handling must be abandoned [§15.6.1]. 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 current *terminate-function* [§17.1.2.1.2].

17.1.2.1.2 *terminate-function*

```
typedef void (*terminate_function)();
```

A C++ program may install *terminate-functions* via calls to `set_terminate()` [§17.1.2.1.3].

Result semantics:

This function shall not return.
It may call `abort()` [§17.4.10.4.1] or `exit()` [§17.4.10.4.3].
Any other action is ***undefined***.

❑ It may re-start the application process, invoke some other last-chance disaster-recovery mechanism, or take some other action that cannot be specified in the standard.

Description of default implementation:

The default *terminate-function* is `abort()` [§17.4.10.4.1].

17.1.2.1.3 `set_terminate()`

```
terminate_function set_terminate( terminate_function );
```

Installs the function given as an argument as the current *terminate-function* [§17.1.2.1.2].

Returns the previous function given to `set_terminate()`.

❑ This enables callers to implement a stack strategy for using *terminate-functions.*

17.1.2.2 `Violating` ***`exception-specifications`***

These functions allow C++ programs to control how the implementation responds to inconsistencies between declared *exception-specifications* and actual exceptions detected at runtime.

17.1.2.2.1 `unexpected()`

```
void unexpected()
```

The implementation calls `unexpected()` if a function with an *exception-specification* throws an exception that is not listed in the *exception-specification*.

`unexpected()` calls the current *unexpected-function* [§17.1.2.2.2]

17.1.2.2.2 *unexpected-function*

```
typedef void (*unexpected_function)();
```

A C++ program may install *unexpected-function*s via calls to `set_unexpected()` [§17.1.2.2.3].

Result semantics:

This function shall not return.
It may call `terminate()` [§17.1.2.1.1], `abort()` [§17.4.10.4.1], or `exit()` [§17.4.10.4.3].
Any other action is ***undefined***.

- ❑ Since these kinds of errors usually indicate design problems that need to be fixed, the intent is to make them easy to detect.

Description of default implementation:

The default *unexpected-function* is `terminate()` [§17.1.2.1.1].

17.1.2.2.3 `set_unexpected()`

```
unexpected_function set_unexpected( unexpected_function );
```

Installs the function given as an argument as the current *unexpected-function* [§17.1.2.2.2].

Returns the previous function given to `set_unexpected()`.

- ❑ This enables callers to implement a stack strategy for using *unexpected-functions*.

17.1.2.3 Predefined exceptions

These classes define the exceptions reported by various functions in the standard library.

17.1.2.3.1 xmsg exception

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

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

❑ 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;
  }
}
```

x.why() is the string used to construct an xmsg x.
That is: **xmsg(s).why() == s.**

❑ The absence of a default constructor means that every xmsg must contain a meaningful message.

xmsg::raise() is defined by:

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

❑ xmsg::raise() adds no functionality but is included as a convenient hook for debugging.

17.1.2.3.2 xalloc exception

```
class xalloc : public xmsg {
public:
  xalloc(const string& msg, size_t requested_size);

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

An xalloc exception can be thrown by the *new-handler* [§17.1.1.4] when it cannot find storage to allocate.

❑ 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.