The default constructor and destructor for new are:

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

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

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

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

How it should be applied in general:

#ifdef \_DEBUG

#define \_DEBUG\_NEW\_REDEFINE\_NEW 0

#include "debug\_new.h"

#else

#define DEBUG\_NEW **new**

#endif

This way if \_DEBUG is defined, the new will be overridden with DEBUG\_NEW. If it \_DEBUG is not defined, DEBUG\_NEW will be defined as the standard new.

**#define new DEBUG\_NEW**

converts **new** into **new(size\_t t, \_\_FILE\_\_, \_\_LINE\_\_)**

file and line are preprocessor macros for automatically getting the name of the file and the line in that file where the **new** is called

**\_\_FILE\_\_** is a macro for char(name of the file)

**\_\_LINE\_\_** is a macro for int (line in file)

Basically I will need to keep count of the newed and deleted objects with a static class:

\*When new is called, take count

\*When delete is called, reduce the count

\*\*Identifying every object would be optimal, can get heavy.

\*Initialized on runtime as it is static

\*Destructor called after the program has exited. Check if there are any unhandled counts.

\*If the count is not 0, then report all the unhandled allocations.

**Used in the debug\_new project:**

**Summarized and discussed**

The operator overloading:

* operator new (size\_t, const char \*, int)
* operator new [] (size\_t, const char \*, int)
* operator new (size\_t)
* operator new [] (size\_t)
* operator new (size\_t, const std:: nothrow\_t &)
* operator new [] (size\_t, const std:: nothrow\_t &)
* operator delete (void \*)
* operator delete [] (void \*)
* operator delete (void \*, const char \*, int)
* operator delete [] (void \*, const char \*, int)
* operator delete (void \*, const std:: nothrow\_t &)
* operator delete [] (void \*, const std:: nothrow\_t &)

Function provided by:

* check\_leaks ()   
  Check for memory leaks occur

Provided by global variables

* new\_verbose\_flag   
  New and delete in the "wordy" to display information
* new\_autocheck\_flag   
  Whether to withdraw from the process is automatic detection of a memory leak

Re-definition of the macro:

* NO\_PLACEMENT\_DELETE   
  Compiler does not support the assumption that placement delete (overall effective)
* DEBUG\_NEW\_NO\_NEW\_REDEFINITION   
  The definition does not re-new, the assumption that users will use their own debug\_new (includes effective debug\_new.h)
* DEBUG\_NEW\_EMULATE\_MALLOC   
  Re-definition of malloc / free, use new / delete simulation (including debug\_new.h the time being in force)
* DEBUG\_NEW\_HASH   
  List to change the hash value memory block algorithm (debug\_new.cpp compile time being in force)
* DEBUG\_NEW\_HASHTABLE\_SIZE   
  Hash list to change the memory block the size of barrels per day (effective debug\_new.cpp compiler)
* DEBUG\_NEW\_FILENAME\_LEN   
  If in the allocation of memory, then copy the file name, file name length of reservations; to 0 automatically when the definition of DEBUG\_NEW\_NO\_FILENAME\_COPY (compile time being in force debug\_new.cpp; see document in the Notes)
* DEBUG\_NEW\_NO\_FILENAME\_COPY   
  Allocate memory for file name is not copied, but only to preserve the pointer; efficient (effective debug\_new.cpp compiler; see document in the Notes)