NAME

libmpatrol − dynamic memory allocation and tracing library

SYNOPSIS

```
#include <mpatrol.h>

void *malloc(size_t size);
void *calloc(size_t nelem, size_t size);
void *memalign(size_t align, size_t size);
void *valloc(size_t size);
void *pvalloc(size_t size);
void *alloca(size_t size);
char *strdup(const char *str);
char *strndup(const char *str, size_t size);
char *strsave(const char *str);
char *strnsave(const char *str, size_t size);
char *strdupa(const char *str);
char *strndupa(const char *str, size_t size);
void *realloc(void *ptr, size_t size);
void *reallocf(void *ptr, size_t size);
void *recalloc(void *ptr, size_t nelem, size_t size);
void *expand(void *ptr, size_t size);
void free(void *ptr);
void cfree(void *ptr, size_t nelem, size_t size);
void dealloca(void *ptr);

void *xmalloc(size_t size);
void *xcalloc(size_t nelem, size_t size);
char *xstrdup(const char *str);
void *xrealloc(void *ptr, size_t size);
void xfree(void *ptr);

void *operator new(size_t size) throw(std::bad_alloc);
void *operator new(size_t size, const std::nothrow_t&) throw();
void *operator new[](size_t size) throw(std::bad_alloc);
void *operator new[](size_t size, const std::nothrow_t&) throw();
void operator delete(void *ptr) throw();
void operator delete(void *ptr, const std::nothrow_t&) throw();
void operator delete[](void *ptr) throw();
void operator delete[](void *ptr, const std::nothrow_t&) throw();
std::new_handler std::set_new_handler(std::new_handler func) throw();

void *memset(void *ptr, int byte, size_t size);
void bzero(void *ptr, size_t size);
void *memccpy(void *dest, const void *src, int byte, size_t size);
void *memcpy(void *dest, const void *src, size_t size);
void *memmove(void *dest, const void *src, size_t size);
void bcopy(const void *src, void *dest, size_t size);
int memcmp(const void *ptr1, const void *ptr2, size_t size);
int bcmp(const void *ptr1, const void *ptr2, size_t size);
void *memchr(const void *ptr, int byte, size_t size);
void *memmem(const void *ptr1, size_t size1, const void *ptr2, size_t size2);

int __mp_atexit(void (*func)(void));
unsigned long __mp_setoption(long opt, unsigned long val);
```

```
int __mp_getoption(long opt, unsigned long *val);
unsigned long __mp_libversion(void);
const char *__mp_strerror(__mp_errortype err);
const char *__mp_function(__mp_alloctype func);
int __mp_setuser(const void *ptr, const void *data);
int __mp_setmark(const void *ptr);
int __mp_info(const void *ptr, __mp_allocinfo *info);
int __mp_syminfo(const void *ptr, __mp_symbolinfo *info);
const char *__mp_symbol(const void *ptr);
int __mp_printinfo(const void *ptr);
unsigned long __mp_snapshot(void);
size_t __mp_iterate(int (*func)(const void *, void *), void *data, unsigned long event);
size_t __mp_iterateall(int (*func)(const void *, void *), void *data);
int __mp_addallocentry(const char *file, unsigned long line, size_t size);
int __mp_addfreeentry(const char *file, unsigned long line, size_t size);
void __mp_clearleaktable(void);
int __mp_startleaktable(void);
int __mp_stopleaktable(void);
void __mp_leaktable(size_t size, int opt, unsigned char flags);
void __mp_memorymap(int stats);
void __mp_summary(void);
int __mp_stats(__mp_heapinfo *info);
void __mp_check(void);
__mp_prologuehandler __mp_prologue(const __mp_prologuehandler);
__mp_epiloguehandler __mp_epilogue(const __mp_epiloguehandler);
__mp_nomemoryhandler __mp_nomemory(const __mp_nomemoryhandler);
int __mp_printf(const char *fmt, ...);
int __mp_vprintf(const char *fmt, va_list args);
void __mp_locprintf(const char *fmt, ...);
void __mp_vlocprintf(const char *fmt, va_list args);
void __mp_logmemory(const void *ptr, size_t size);
int __mp_logstack(size_t frames);
int __mp_logaddr(const void *ptr);
int __mp_edit(const char *file, unsigned long line);
int __mp_list(const char *file, unsigned long line);
int __mp_view(const char *file, unsigned long line);
int __mp_readcontents(const char *file, void *ptr);
int __mp_writecontents(const char *file, const void *ptr);
long __mp_cmpcontents(const char *file, const void *ptr);
int __mp_remcontents(const char *file, const void *ptr);

__mp_errortype __mp_errno;
```

DESCRIPTION

The *mpatrol library* contains implementations of dynamic memory allocation functions for C and C++ suitable for tracing and debugging, and is available on UNIX, AmigaOS, Windows and Netware platforms. The library is intended to be used without requiring any changes to existing user source code except the inclusion of the *mpatrol.h* header file, although additional functions are supplied for extra tracing and control. Note that the current version of the mpatrol library is contained in the MPATROL_VERSION preprocessor macro.

All of the function definitions in *mpatrol.h* can be disabled by defining the NDEBUG preprocessor macro, which is the same macro used to control the behaviour of the assert function. If NDEBUG is defined then no macro redefinition of functions will take place and all special mpatrol library functions will evaluate to empty statements. The *mpalloc.h* header file will also be included in this case. It is intended that the NDEBUG preprocessor macro be defined in release builds.

The MP_MALLOC family of functions that are defined in *mpalloc.h* are also defined in *mpatrol.h* when NDEBUG is not defined. The mpatrol versions of these functions contain more debugging information than the mpalloc versions do, but they do not call the allocation failure handler when no more memory is available (they cause the *OUTMEM* error message to be given instead). See mpalloc(3) for the descriptions of the MP_MALLOC family of functions.

All diagnostics are sent to the file *mpatrol.log* in the current directory by default but this can be changed at run-time. Additional configuration options can also be changed at run-time by setting and altering the MPATROL_OPTIONS environment variable. In addition, the LOGFILE, PROFFILE and TRACEFILE options are affected by the LOGDIR, PROFDIR and TRACEDIR environment variables respectively. See ENVIRONMENT below for more details.

Details of memory allocations and free memory are stored internally as a tree structure for speed and also to allow the best fit allocation algorithm to be used. This also enables the library to perform intelligent resizing of memory allocations and can be used to quickly determine if an address has been allocated on the heap.

On systems that support memory protection, the library attempts to detect any illegal memory accesses and display as much information as it can obtain about the address in question and where the illegal memory access occurred.

Stack traceback information for every memory allocation is available on some supported platforms, which is useful for determining exactly where a memory allocation was performed or for adding meaning to tracing. Symbol names are read from the executable file and also possibly from any required shared libraries, and if the USEDEBUG option is used and is available then the debugging section in the executable file will be read to determine additional source-level information.

On systems that support it, global functions (with C linkage) in an executable file or shared library whose names begin with __mp_init_ will be noted when the mpatrol library first starts up and is reading the symbols. Such functions will then be called as soon as the mpatrol library is initialised, which can be useful if the initialisation occurs before main is called. These functions must accept no arguments and must return no value. Similar behaviour exists for global functions whose names begin with __mp_fini_, except that such functions will be executed when the mpatrol library terminates. Note that this feature will have no effect if the symbol table is stripped from the executable file or shared library before the program is run, and the order in which such functions will be called if there are more than one is unspecified.

On UNIX platforms, the fork function can cause problems if it is used to make a copy of the parent process without immediately calling one of the exec family of functions. This is because the child process inherits all of the memory allocations of the parent process, but also inherits the log, profile and trace files as well. If both the parent and child processes make subsequent memory allocations there will be multiple entries with the same allocation indices written to the log, profile or trace files. This can be most confusing when processing these files afterwards! As a workaround, the mpatrol library will always check the current process identifier every time one of its functions is called if the CHECKFORK option is used and will open new log, profile or trace files if it has determined that the process has been forked. If the CHECKFORK option is not used then a call to __mp_reinit should be added as the first function call in the child process in order to duplicate the behaviour of the CHECKFORK option.

Memory allocation profiling is supported, with statistics about every memory allocation and deallocation that was made during the execution of a program being written to a file at program termination if the PROF option is used. The information stored in this file can then be used by the mprof command to display various tables summarising the memory allocation behaviour of the program that produced it. Memory allocation tracing is also supported, where a trace of all memory allocations, reallocations and deallocations can be written to a tracing output file in a concise encoded format for later processing by the mptrace command. This is controlled with the TRACE option.

FUNCTIONS

The following 19 functions are available as replacements for existing C library functions. To use these you must include *mpatrol.h* before all other header files, although on UNIX and Windows platforms (and AmigaOS when using gcc) they will be used anyway, albeit with slightly less tracing information. If alloca is

being used and *alloca.h* is included then *mpatrol.h* must appear after *alloca.h* otherwise the debugging version of alloca will not be used:

malloc Allocates *size* uninitialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *size* bytes in length. If *size* is *0* then the memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc.

calloc Allocates *nelem* elements of *size* zero-initialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *nelem * size* bytes in length. If *nelem * size* is *0* then the amount of memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc.

memalign
Allocates *size* uninitialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be aligned to *align* bytes and can be used to store data of up to *size* bytes in length. If *align* is zero then the default system alignment will be used. If *align* is not a power of two then it will be rounded up to the nearest power of two. If *align* is greater than the system page size then it will be truncated to that value. If *size* is *0* then the memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc, although the latter will not guarantee the preservation of alignment.

valloc Allocates *size* uninitialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be aligned to the system page size and can be used to store data of up to *size* bytes in length. If *size* is *0* then the memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc, although the latter will not guarantee the preservation of alignment.

pvalloc Allocates *size* uninitialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be aligned to the system page size and can be used to store data of up to *size* bytes in length. If *size* is *0* then the memory allocated will be implicitly rounded up to *1* page, otherwise *size* will be implicitly rounded up to a multiple of the system page size. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc, although the latter will not guarantee the preservation of alignment.

alloca Allocates *size* temporary uninitialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *size* bytes in length. If *size* is *0* then the memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then the program will be terminated and the *OUTMEM* error will be given. The alloca function normally allocates its memory from the stack, with the result that all such allocations will be freed when the function returns. This version of alloca allocates its memory from the heap in order to provide better debugging, but the allocations may not necessarily be freed immediately when the function returns. The allocated memory can be deallocated explicitly with dealloca, but may not be reallocated or deallocated in any other way. This function is available for backwards compatibility with older C source code and should not be used in new code.

strdup Allocates exactly enough memory from the heap to duplicate *str* (including the terminating nul character) and returns a pointer to the first byte of the allocation after copying *str* to the newly-allocated memory. The pointer returned will have no alignment constraints and can be used to store character data up to the length of *str*. If *str* is NULL then an error will be given and the null

pointer will be returned. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc.

strndup

Allocates exactly enough memory from the heap to duplicate *str* (including the terminating nul character) and returns a pointer to the first byte of the allocation after copying *str* to the newly-allocated memory. The pointer returned will have no alignment constraints and can be used to store character data up to the length of *str*. If *str* is NULL and *size* is non-zero then an error will be given and the null pointer will be returned. If the length of *str* is greater than *size* then only *size* characters will be allocated and copied, with one additional byte for the nul character. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc. This function is available for backwards compatibility with older C libraries and should not be used in new code.

strsave Allocates exactly enough memory from the heap to duplicate *str* (including the terminating nul character) and returns a pointer to the first byte of the allocation after copying *str* to the newly-allocated memory. The pointer returned will have no alignment constraints and can be used to store character data up to the length of *str*. If *str* is NULL then an error will be given and the null pointer will be returned. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc. This function is available for backwards compatibility with older C libraries and should not be used in new code.

strnsave

Allocates exactly enough memory from the heap to duplicate *str* (including the terminating nul character) and returns a pointer to the first byte of the allocation after copying *str* to the newly-allocated memory. The pointer returned will have no alignment constraints and can be used to store character data up to the length of *str*. If *str* is NULL and *size* is non-zero then an error will be given and the null pointer will be returned. If the length of *str* is greater than *size* then only *size* characters will be allocated and copied, with one additional byte for the nul character. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free or reallocated with realloc. This function is available for backwards compatibility with older C libraries and should not be used in new code.

strdupa

Allocates exactly enough temporary memory from the heap to duplicate *str* (including the terminating nul character) and returns a pointer to the first byte of the allocation after copying *str* to the newly-allocated memory. The pointer returned will have no alignment constraints and can be used to store character data up to the length of *str*. If *str* is NULL then an error will be given and the null pointer will be returned. If there is not enough space in the heap then the program will be terminated and the *OUTMEM* error will be given. The strdupa function normally allocates its memory from the stack, with the result that all such allocations will be freed when the function returns. This version of strdupa allocates its memory from the heap in order to provide better debugging, but the allocations may not necessarily be freed immediately when the function returns. The allocated memory can be deallocated explicitly with dealloca, but may not be reallocated or deallocated in any other way. This function is available for backwards compatibility with older C source code and should not be used in new code.

strndupa

Allocates exactly enough temporary memory from the heap to duplicate *str* (including the terminating nul character) and returns a pointer to the first byte of the allocation after copying *str* to the newly-allocated memory. The pointer returned will have no alignment constraints and can be used to store character data up to the length of *str*. If *str* is NULL and *size* is non-zero then an error will be given and the null pointer will be returned. If the length of *str* is greater than *size* then only

size characters will be allocated and copied, with one additional byte for the nul character. If there is not enough space in the heap then the program will be terminated and the *OUTMEM* error will be given. The strndupa function normally allocates its memory from the stack, with the result that all such allocations will be freed when the function returns. This version of strndupa allocates its memory from the heap in order to provide better debugging, but the allocations may not necessarily be freed immediately when the function returns. The allocated memory can be deallocated explicitly with dealloca, but may not be reallocated or deallocated in any other way. This function is available for backwards compatibility with older C source code and should not be used in new code.

realloc Resizes the memory allocation beginning at *ptr* to *size* bytes and returns a pointer to the first byte of the new allocation after copying *ptr* to the newly-allocated memory, which will be truncated if *size* is smaller than the original allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *size* bytes in length. If *ptr* is NULL then the call will be equivalent to malloc. If *size* is *0* then the existing memory allocation will be freed and the null pointer will be returned. If *size* is greater than the original allocation then the extra space will be filled with uninitialised bytes. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free and can be reallocated again with realloc.

reallocf

Resizes the memory allocation beginning at *ptr* to *size* bytes and returns a pointer to the first byte of the new allocation after copying *ptr* to the newly-allocated memory, which will be truncated if *size* is smaller than the original allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *size* bytes in length. If *ptr* is NULL then the call will be equivalent to malloc. If *size* is *0* then the existing memory allocation will be freed and the null pointer will be returned. If *size* is greater than the original allocation then the extra space will be filled with uninitialised bytes. If there is not enough space in the heap then the null pointer will be returned, the original allocation will be freed and errno will be set to ENOMEM. The allocated memory must be deallocated with free and can be reallocated again with realloc. This function is available for backwards compatibility with older C libraries and should not be used in new code.

recalloc

Resizes the memory allocation beginning at *ptr* to *nelem* elements of *size* bytes and returns a pointer to the first byte of the new allocation after copying *ptr* to the newly-allocated memory, which will be truncated if *nelem* * *size* is smaller than the original allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *nelem* * *size* bytes in length. If *ptr* is NULL then the call will be equivalent to calloc. If *nelem* * *size* is *0* then the existing memory allocation will be freed and the null pointer will be returned. If *nelem* * *size* is greater than the original allocation then the extra space will be filled with zero-initialised bytes. If there is not enough space in the heap then the null pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free and can be reallocated again with realloc. This function is available for backwards compatibility with older C libraries and calloc and should not be used in new code.

expand

Attempts to resize the memory allocation beginning at *ptr* to *size* bytes and either returns *ptr* if there was enough space to resize it, or NULL if the block could not be resized for a particular reason. If *ptr* is NULL then the call will be equivalent to malloc. If *size* is 0 then the existing memory allocation will be freed and the NULL pointer will be returned. If *size* is greater than the original allocation then the extra space will be filled with uninitialised bytes and if *size* is less than the original allocation then the memory block will be truncated. If there is not enough space in the heap then the NULL pointer will be returned and errno will be set to ENOMEM. The allocated memory must be deallocated with free and can be reallocated again with realloc. This function is available for backwards compatibility with older C libraries and should not be used in new code.

free Frees the memory allocation beginning at *ptr* so the memory can be reused by another call to allocate memory. If *ptr* is NULL then no memory will be freed. All of the previous contents will be destroyed.

cfree Frees the memory allocation beginning at *ptr* so the memory can be reused by another call to allocate memory. If *ptr* is NULL then no memory will be freed. All of the previous contents will be destroyed. The *nelem* and *size* parameters are ignored in this implementation. This function is available for backwards compatibility with older C libraries and calloc and should not be used in new code.

dealloca

Explicitly frees the temporary memory allocation beginning at *ptr* so the memory can be reused by another call to allocate memory. If *ptr* is NULL then no memory will be explicitly freed. All of the previous contents will be destroyed. This function can only be used to free memory that was allocated with the alloca, strdupa and strndupa functions, but is only really required if the mpatrol library does not automatically free such memory allocations when the allocating function returns. This function is mpatrol-specific and should not be used in release code.

The following 5 functions are available as replacements for existing C library extension functions that always abort and never return NULL if there is insufficient memory to fulfil a request. To use these you must include *mpatrol.h* before all other header files, although on UNIX and Windows platforms (and AmigaOS when using gcc) they will be used anyway, albeit with slightly less tracing information:

xmalloc

Allocates *size* uninitialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *size* bytes in length. If *size* is *0* then the memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then the program will be terminated and the *OUTMEM* error will be given. The allocated memory must be deallocated with xfree or reallocated with xrealloc.

xcalloc Allocates *nelem* elements of *size* zero-initialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *nelem * size* bytes in length. If *nelem * size* is *0* then the amount of memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then the program will be terminated and the *OUTMEM* error will be given. The allocated memory must be deallocated with xfree or reallocated with xrealloc.

xstrdup

Allocates exactly enough memory from the heap to duplicate *str* (including the terminating nul character) and returns a pointer to the first byte of the allocation after copying *str* to the newly-allocated memory. The pointer returned will have no alignment constraints and can be used to store character data up to the length of *str*. If *str* is NULL then an error will be given and the null pointer will be returned. If there is not enough space in the heap then the program will be terminated and the *OUTMEM* error will be given. The allocated memory must be deallocated with xfree or reallocated with xrealloc.

xrealloc

Resizes the memory allocation beginning at *ptr* to *size* bytes and returns a pointer to the first byte of the new allocation after copying *ptr* to the newly-allocated memory, which will be truncated if *size* is smaller than the original allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *size* bytes in length. If *ptr* is NULL then the call will be equivalent to xmalloc. If *size* is *0* then it will be implictly rounded up to *1*. If *size* is greater than the original allocation then the extra space will be filled with uninitialised bytes. If there is not enough space in the heap then the program will be terminated and the *OUTMEM* error will be given. The allocated memory must be deallocated with xfree and can be reallocated again with xrealloc.

xfree Frees the memory allocation beginning at *ptr* so the memory can be reused by another call to allocate memory. If *ptr* is NULL then no memory will be freed. All of the previous contents will be destroyed.

The following 5 functions are available as replacements for existing C++ library functions, but the replacements in *mpatrol.h* will only be used if the MP_NOCPLUSPLUS preprocessor macro is not defined. The replacement operators make use of the preprocessor in order to obtain source-level information. If this causes problems then you should define the MP_NONEWDELETE preprocessor macro and use the MP_NEW, MP_NEW_NOTHROW and MP_DELETE macros instead of new and delete directly. To use these C++ features you must include *mpatrol.h* before all other header files, although on UNIX and Windows platforms (and AmigaOS when using gcc) they will be used anyway, albeit with slightly less tracing information:

operator new
: Allocates *size* uninitialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *size* bytes in length. If *size* is *0* then the memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then either the *std::bad_alloc* exception will be thrown or the null pointer will be returned and errno will be set to ENOMEM - the behaviour depends on whether the nothrow version of the operator is used. The allocated memory must be deallocated with operator delete.

operator new[]
: Allocates *size* uninitialised bytes from the heap and returns a pointer to the first byte of the allocation. The pointer returned will be suitably aligned for casting to any type and can be used to store data of up to *size* bytes in length. If *size* is *0* then the memory allocated will be implicitly rounded up to *1* byte. If there is not enough space in the heap then either the *std::bad_alloc* exception will be thrown or the null pointer will be returned and errno will be set to ENOMEM - the behaviour depends on whether the nothrow version of the operator is used. The allocated memory must be deallocated with operator delete[].

operator delete
: Frees the memory allocation beginning at *ptr* so the memory can be reused by another call to allocate memory. If *ptr* is NULL then no memory will be freed. All of the previous contents will be destroyed. This function must only be used with memory allocated by operator new.

operator delete[]
: Frees the memory allocation beginning at *ptr* so the memory can be reused by another call to allocate memory. If *ptr* is NULL then no memory will be freed. All of the previous contents will be destroyed. This function must only be used with memory allocated by operator new[].

set_new_handler
: Installs a low-memory handler specifically for use with operator new and operator new[] and returns a pointer to the previously installed handler, or the null pointer if no handler had been previously installed. This will be called repeatedly by both functions when they would normally return NULL, and this loop will continue until they manage to allocate the requested space. Note that this function is equivalent to __mp_nomemory and will replace the handler installed by that function.

The following 10 functions are available as replacements for existing C library memory operation functions. To use these you must include *mpatrol.h* before all other header files, although on UNIX and Windows platforms (and AmigaOS when using gcc) they will be used anyway, albeit with slightly less tracing information:

memset
: Writes *size* bytes of value *byte* to the memory location beginning at *ptr* and returns *ptr*. If *size* is 0 then no bytes will be written. If the operation would affect an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be written.

bzero Writes *size* zero bytes to the memory location beginning at *ptr*. If *size* is 0 then no bytes will be written. If the operation would affect an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be written. This function is available for backwards compatibility with older C libraries and should not be used in new code.

memccpy
Copies *size* bytes from *src* to *dest* and returns NULL, or copies the number of bytes up to and including the first occurrence of *byte* if *byte* exists within the specified range and returns a pointer to the first byte after *byte*. If *size* is 0 or *src* is the same as *dest* then no bytes will be copied. The source and destination ranges should not overlap, otherwise a warning will be written to the log file. If the operation would affect an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be copied.

memcpy
Copies *size* bytes from *src* to *dest* and returns *dest*. If *size* is 0 or *src* is the same as *dest* then no bytes will be copied. The source and destination ranges should not overlap, otherwise a warning will be written to the log file. If the operation would affect an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be copied.

memmove
Copies *size* bytes from *src* to *dest* and returns *dest*. If *size* is 0 or *src* is the same as *dest* then no bytes will be copied. If the operation would affect an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be copied.

bcopy Copies *size* bytes from *src* to *dest*. If *size* is 0 or *src* is the same as *dest* then no bytes will be copied. If the operation would affect an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be copied. This function is available for backwards compatibility with older C libraries and should not be used in new code.

memcmp
Compares *size* bytes from *ptr1* and *ptr2* and returns 0 if all of the bytes are identical, or returns the byte difference of the first differing bytes. If *size* is 0 or *ptr1* is the same as *ptr2* then no bytes will be compared. If the operation would read from an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be compared.

bcmp Compares *size* bytes from *ptr1* and *ptr2* and returns 0 if all of the bytes are identical, or returns the byte difference of the first differing bytes. If *size* is 0 or *ptr1* is the same as *ptr2* then no bytes will be compared. If the operation would read from an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be compared. This function is available for backwards compatibility with older C libraries and should not be used in new code.

memchr
Searches up to *size* bytes in *ptr* for the first occurrence of *byte* and returns a pointer to it or NULL if no such byte occurs. If *size* is 0 then no bytes will be searched. If the operation would affect an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in the log file and no bytes will be searched.

memmem
Searches up to *size1* bytes in *ptr1* for the first occurrence of *ptr2* (which is exactly *size2* bytes in length) and returns a pointer to it or NULL if no such sequence of bytes occur. If *size1* or *size2* is 0 then no bytes will be searched. If the operation would affect an existing memory allocation in the heap but would straddle that allocation's boundaries then an error message will be generated in

the log file and no bytes will be searched.

The following 42 functions are available as support routines for additional control and tracing in the mpatrol library. Although they are documented here as being prefixed by __mp_, their equivalent functions that are prefixed by mpatrol_ are also defined as aliases in the *mpatrol.h* header file. To use these you should include the *mpatrol.h* header file:

__mp_atexit
: Installs a function to be called when the mpatrol library terminates. Up to 32 such functions can be registered and will be called in reverse order of registration. Returns *1* on success or *0* if *func* could not be registered.

__mp_setoption
: Sets the value of an mpatrol option after the library has been initialised. Options that require values are listed in *mpatrol.h* prefixed with *MP_OPT_**. The *opt* argument should be set to one of these macros, and the *val* argument should be set to the option value, cast to an unsigned integer. The return value will be *0* on success and *1* on failure. Options that are flags are listed in *mpatrol.h* prefixed with *MP_FLG_**. Multiple flags can be set or unset at once using the *MP_OPT_SETFLAGS* and *MP_OPT_UNSETFLAGS* options respectively, with the necessary flags specified in *val*. The return value will be *0* on success and a combination of all of the flags that could not be set or unset on failure.

__mp_getoption
: Gets the value of an mpatrol option after the library has been initialised. If *opt* is a valid option listed in *mpatrol.h* then *1* will be returned and the associated value will be returned in *val* and cast to an unsigned integer, otherwise *0* will be returned. If *opt* is *MP_OPT_SETFLAGS* then all of the mpatrol library flags that are set will be returned in *val*. If *opt* is *MP_OPT_UNSETFLAGS* then all of the mpatrol library flags that are not set will be returned in *val*.

__mp_libversion
: Returns the version number of the mpatrol library. This can be useful for verifying that the version of the mpatrol library that a program is linked with is the one expected at compile-time.

__mp_strerror
: Returns the error message corresponding to the error code *err* or NULL if no such error code exists. The most recent error code recorded by the mpatrol library can be obtained by examining __mp_errno.

__mp_function
: Returns the name of the function corresponding to the allocation type *func* or NULL if no such allocation type exists.

__mp_setuser
: Sets the user data for the memory allocation containing *ptr*. The contents of *data* are entirely application-specific as user data will never be examined by the mpatrol library. Such data is associated with a memory allocation for its entire lifetime unless overridden by a subsequent call to __mp_setuser. As such, the user data must be valid for the entire lifetime of the memory allocation, perhaps even after the allocation has been freed if the NOFREE option is being used. This function returns *1* if there is an allocated memory block containing *ptr*, and *0* otherwise.

__mp_setmark
: Sets the marked flag for the memory allocation containing *ptr*, indicating that the memory allocation cannot be freed (but can be reallocated) and thus will not be listed as a memory leak. This function returns *1* if there is an allocated memory block containing *ptr*, and *0* otherwise. Note that a memory allocation made by alloca, strdupa or strndupa may not be marked.

__mp_info
: Obtains information about a specific memory allocation by placing statistics about *ptr* in *info*. If *ptr* does not belong to a previously allocated memory allocation or free memory block then *0* will be returned, otherwise *1* will be returned and *info* will contain the following information (note that

a free memory block will only contain the *block* and *size* fields and can be identified by not having the *allocated* flag set):

Field	Description
block	Pointer to first byte of alloc.
size	Size of alloc in bytes.
type	Type of function which allocated memory.
alloc	Allocation index.
realloc	Number of times reallocated.
thread	Thread identifier.
event	Event of last modification.
func	Function in which alloc took place.
file	File in which alloc took place.
line	Line number at which alloc took place.
stack	Pointer to function call stack.
typestr	Type stored in allocation.
typesize	Size of type stored in allocation.
userdata	User data associated with allocation.
allocated	Indicates if alloc was allocated.
freed	Indicates if alloc has been freed.
marked	Indicates if alloc has been marked.
profiled	Indicates if alloc has been profiled.
traced	Indicates if alloc has been traced.
internal	Indicates if alloc is internal.

__mp_syminfo

Obtains symbolic information about a specific code address by placing statistics about *ptr* in *info*. If *ptr* does not belong to a function symbol then *0* will be returned, otherwise *1* will be returned and *info* will contain the following information:

Field	Description
name	Name of symbol.
object	File containing symbol.
addr	Start address of symbol.
size	Size of symbol.
file	Filename corresponding to address.
line	Line number corresponding to address.

__mp_symbol

Obtains the name of a function symbol containing the code address specified in *ptr*. If *ptr* does not belong to a function symbol then NULL will be returned.

__mp_printinfo

Displays information about a specific memory allocation containing *ptr* to the standard error file stream. If *ptr* does not belong to a previously allocated memory allocation or free memory block then *0* will be returned, otherwise *1* will be returned. This function is intended to be called from within a debugger.

__mp_snapshot

Returns the current event number, effectively taking a snapshot of the heap. This number can then be used in later calls to __mp_iterate.

__mp_iterate

Iterates over all of the current allocated and freed memory allocations, calling *func* with the start address of every memory allocation that has been modified since event number *event*. If *func* is

NULL then __mp_printinfo will be used as the callback function. If event is *0* then *func* will be called with the start address of every memory allocation. If *func* returns a negative number then the iteration process will be stopped immediately. If *func* returns a positive number above zero then __mp_iterate will return the number of times *func* returned a non-zero number after the iteration process has stopped. The *data* argument is passed directly to *func* as its second argument and is not read by the mpatrol library.

__mp_iterateall
: Iterates over all of the current allocated and freed memory allocations and any free memory blocks, calling *func* with the start address of every memory allocation or free block. If *func* is NULL then __mp_printinfo will be used as the callback function. If *func* returns a negative number then the iteration process will be stopped immediately. If *func* returns a positive number above zero then __mp_iterate will return the number of times *func* returned a non-zero number after the iteration process has stopped. The *data* argument is passed directly to *func* as its second argument and is not read by the mpatrol library. Note that unlike __mp_iterate, this function will also include internal memory allocations made by the mpatrol library and is intended for walking the entire heap.

__mp_addallocentry
: Adds an entry representing an allocation of size *size* to the leak table. The allocation will be associated with a source filename of *file* and a line number of *line* if the former is non-NULL and the latter is non-zero. If *file* is non-NULL and *line* is *0* then *file* represents the name of the function that made the allocation. If *file* is NULL and *line* is non-zero then *line* represents the code address at which the allocation was made. If *file* is NULL and *line* is *0* then the location of the allocation is unknown. Returns *1* on success and *0* if there was no more memory available to add another entry to the leak table.

__mp_addfreeentry
: Adds an entry representing a deallocation of size *size* to the leak table. The deallocation will be associated with a source filename of *file* and a line number of *line* if the former is non-NULL and the latter is non-zero. If *file* is non-NULL and *line* is *0* then *file* represents the name of the function that made the deallocation. If *file* is NULL and *line* is non-zero then *line* represents the code address at which the deallocation was made. If *file* is NULL and *line* is *0* then the location of the deallocation is unknown. Returns *1* on success and *0* if there was no existing allocation from the same location in the leak table.

__mp_clearleaktable
: Deletes all of the existing entries in the leak table, making it empty. This will also affect the behaviour of the LEAKTABLE option since that option will then only be able to show a summary of the entries in the leak table that were collected after the last call to this function rather than from the start of program execution.

__mp_startleaktable
: Starts the automatic logging of all memory allocations, reallocations and deallocations to the leak table. Returns *1* if such logging was already being performed and *0* otherwise.

__mp_stopleaktable
: Stops the automatic logging of all memory allocations, reallocations and deallocations to the leak table. Returns *1* if such logging was already being performed and *0* otherwise.

__mp_leaktable
: Displays a summary of up to *size* entries from the leak table, or all entries if *size* is *0*. If *opt* is *MP_LT_ALLOCATED* then all allocated entries will be displayed, if *opt* is *MP_LT_FREED* then all freed entries will be displayed and if *opt* is *MP_LT_UNFREED* then all unfreed entries will be displayed. The summary is normally sorted in descending order of total bytes from each entry, but this can be changed by setting *flags* to any combination of *MP_LT_COUNTS* (to sort by the number of occurrences in each entry) and *MP_LT_BOTTOM* (to sort in ascending order).

__mp_memorymap
: If *stats* is non-zero then the current statistics of the mpatrol library will be displayed. If the heap contains at least one allocated, freed or free block then a map of the current heap will also be displayed.

__mp_summary
: Displays information about the current state of the mpatrol library, including its settings and any relevant statistics.

__mp_stats
: Obtains statistics about the current state of the heap and places them in *info*. If this information could not be determined then *0* will be returned, otherwise *1* will be returned and *info* will contain the following information:

Field	Description
acount	Total number of allocated blocks.
atotal	Total size of allocated blocks.
fcount	Total number of free blocks.
ftotal	Total size of free blocks.
gcount	Total number of freed blocks.
gtotal	Total size of freed blocks.
icount	Total number of internal blocks.
itotal	Total size of internal blocks.
mcount	Total number of marked blocks.
mtotal	Total size of marked blocks.

__mp_check
: Forces the library to perform an immediate check of the overflow buffers of every memory allocation and to ensure that nothing has overwritten any free blocks. If any memory allocations made by the alloca family of functions are out of scope then this function will also cause them to be freed.

__mp_prologue
: Installs a prologue function to be called before any memory allocation, reallocation or deallocation function. This function will return a pointer to the previously installed prologue function, or the null pointer if no prologue function had been previously installed. The following arguments will be used to call the prologue function (the last four arguments contain the function name, file name, line number and the return address of the calling function, or null pointers and zero if they cannot be determined):

Argument 1	Argument 2	Argument 3	Called by
-1	*size*	*align*	malloc, etc.
ptr	*size*	*align*	realloc, etc.
ptr	*-1*	*0*	free, etc.
ptr	*-2*	*1*	strdup, etc.

__mp_epilogue
: Installs an epilogue function to be called after any memory allocation, reallocation or deallocation function. This function will return a pointer to the previously installed epilogue function, or the null pointer if no epilogue function had been previously installed. The following arguments will be used to call the epilogue function (the last four arguments contain the function name, file name, line number and the return address of the calling function, or null pointers and zero if they cannot be determined):

Argument	Called by
ptr	malloc, realloc, strdup, etc.
-1	free, etc.

__mp_nomemory
: Installs a low-memory handler and returns a pointer to the previously installed handler, or the null pointer if no handler had been previously installed. This will be called once by C memory allocation functions, and repeatedly by C++ memory allocation functions, when they would normally return NULL. The four arguments contain the function name, file name, line number and the return address of the calling function, or null pointers and zero if they cannot be determined. Note that this function is equivalent to set_new_handler and will replace the handler installed by that function.

__mp_printf
: Writes format string *fmt* with variable arguments to the log file, with each line prefixed by >. The final length of the string that is written to the log file must not exceed 1024 characters. Returns the number of characters written, or a negative number upon error.

__mp_vprintf
: Writes format string *fmt* with variable argument list *args* to the log file, with each line prefixed by >. The final length of the string that is written to the log file must not exceed 1024 characters. Returns the number of characters written, or a negative number upon error.

__mp_locprintf
: Writes format string *fmt* with variable arguments to the log file, with each line prefixed by >. The final length of the string that is written to the log file must not exceed 1024 characters. It also writes information to the log file about where the call to this function was made, which includes the source file location and the call stack if they are available.

__mp_vlocprintf
: Writes format string *fmt* with variable argument list *args* to the log file, with each line prefixed by >. The final length of the string that is written to the log file must not exceed 1024 characters. It also writes information to the log file about where the call to this function was made, which includes the source file location and the call stack if they are available.

__mp_logmemory
: Displays the contents of a block of memory beginning at *ptr*, dumping *size* consecutive bytes to the log file in hexadecimal format.

__mp_logstack
: Displays the current call stack, skipping *frames* stack frames from the current stack frame before writing the symbolic stack trace to the log file. Returns *1* if successful, or *0* if the call stack could not be determined or if *frames* was too large for the current call stack.

__mp_logaddr
: Displays information about a specific memory allocation containing *ptr* to the log file. If *ptr* does not belong to a previously allocated memory allocation then *0* will be returned, otherwise *1* will be returned.

__mp_edit
: Invokes a text editor to edit *file* at line number *line* via the mpedit command. Returns *1* if the text editor was successfully invoked, *-1* if there was an error, or *0* if there is no support for this feature. This function will only work on a system where the EDIT option works.

__mp_list
: Displays a context listing of *file* at line number *line* via the mpedit command. Returns *1* if the listing was successfully performed, *-1* if there was an error, or *0* if there is no support for this feature. This function will only work on a system where the LIST option works.

__mp_view
: Either invokes a text editor to edit *file* at line number *line* or displays a context listing of *file* at line number *line* depending on the setting of the EDIT and LIST options. This is done via the mpedit command and will have no effect if the EDIT and LIST options are not set or if these options are not supported on the system. Returns *1* if the edit or listing was successfully performed, *-1* if there was an error, or *0* if neither of the options were set or if there is no support for this feature.

__mp_readcontents
: Reads the contents of a memory allocation contents file into the memory allocation containing *ptr*. The name of the file is composed of the *file* string followed by the allocation index of the memory allocation separated by a dot. If *file* is NULL then it is assumed to be *0* otherwise.

__mp_writecontents
: Writes the contents of the memory allocation containing *ptr* to an allocation contents file. The name of the file is composed of the *file* string followed by the allocation index of the memory allocation separated by a dot. If *file* is NULL then it is assumed to be *.mpatrol*. Returns *1* if the contents were written successfully and *0* otherwise.

__mp_cmpcontents
: Compares the contents of the memory allocation containing *ptr* with the contents of a previously written allocation contents file. The name of the file is composed of the *file* string followed by the allocation index of the memory allocation separated by a dot. If *file* is NULL then it is assumed to be *.mpatrol*. Any differences are written to the mpatrol log file. Returns the number of differences found, or *-1* if there was an error.

__mp_remcontents
: Removes the memory allocation contents file that corresponds to the memory allocation containing *ptr*. The name of the file is composed of the *file* string followed by the allocation index of the memory allocation separated by a dot. If *file* is NULL then it is assumed to be *0* otherwise.

The following global variable is available for additional control in the mpatrol library. To use it you should include the *mpatrol.h* header file:

__mp_errno
: Contains the most recent error code encountered by the mpatrol library. Its value can be reset to *MP_ET_NONE* before calling an mpatrol library function, and then examined afterwards, either by comparison with the known error codes in the *__mp_errortype* enumeration, or with __mp_strerror.

LINKING

In order to use the mpatrol library on UNIX platforms, the following libraries must be linked in before any other library that defines dynamic memory allocation functions with the same names:

Library	Reason
−lmpatrol	To use this library.
−lmpatrolmt	To use the thread-safe mpatrol library.
−lmpalloc	To use the release library.
−lmptools	To use the mpatrol tools library.
−lld	If built with COFF or XCOFF support.
−lelf	If built with ELF support.
−lbfd & −liberty	If built with BFD support.
−lcl	If built on HP/UX.
−lexc	If built on IRIX or Tru64.
−limagehlp	If built on Windows.
−lpthreads	If built on AIX with threads support.
−lthread	If built on DG/UX with threads support.
−lpthread	If built on UNIX with threads support.

On UNIX platforms, if there were no calls to memory allocation functions before *−lmpatrol* or *−lmpatrolmt* appears on the link line then the mpatrol library will not be linked in if it is an archive library. However, this can be overridden by placing *−umalloc* just before that point.

You may also wish to set your core file size limit to be zero before running any programs linked with the mpatrol library as the extra memory that the library uses can make such files much larger than normal, and if you are planning on using a symbolic debugger then you won't need the core files anyway.

ENVIRONMENT

The library can read certain options at run-time from an environment variable called MPATROL_OPTIONS. This variable must contain one or more valid option keywords from the list below and must be no longer than 1024 characters in length. If MPATROL_OPTIONS is unset or empty then the default settings will be used.

The syntax for options specified within the MPATROL_OPTIONS environment variable is OPTION or OPTION=*VALUE*, where OPTION is a keyword from the list below and *VALUE* is the setting for that option. If *VALUE* is numeric then it may be specified using binary, octal, decimal or hexadecimal notation, with binary notation beginning with either *0b* or *0B*. If *VALUE* is a character string containing spaces then it may be quoted using double quotes. No whitespace may appear between the = sign, but whitespace must appear between different options. Note that option keywords can be given in lowercase as well as uppercase, or a mixture of both.

ALLOCBYTE=*unsigned integer*

Specifies an 8-bit byte pattern with which to prefill newly-allocated memory. This can be used to detect the use of memory which has not been initialised after allocation. Note that this setting will not affect memory allocated with calloc or recalloc as these functions always prefill allocated memory with an 8-bit byte pattern of zero. Default value: ALLOCBYTE=*0xFF*.

ALLOCSTOP=*unsigned integer*

Specifies an allocation index at which to stop the program when it is being allocated. When the number of memory allocations reaches this number the program will be halted, and its state may be examined at that point by using a suitable debugger. Note that this setting will be ignored if its value is zero. Default value: ALLOCSTOP=*0*.

ALLOWOFLOW

Specifies that a warning rather than an error should be produced if any memory operation function overflows the boundaries of a memory allocation, and that the operation should still be performed. This option is provided for circumstances where it is desirable for the memory operation to be performed, regardless of whether it is erroneous or not.

AUTOSAVE=*unsigned integer*

Specifies the frequency at which to periodically write the profiling data to the profiling output file. When the total number of profiled memory allocations and deallocations is a multiple of this number then the current profiling information will be written to the profiling output file. This option can be used to instruct the mpatrol library to dump out any profiling information just before a fatal error occurs in a program, for example. Note that this setting will be ignored if its value is zero. Default value: AUTOSAVE=*0*.

CHECK=*unsigned range*

Specifies a range of allocation indices at which to check the integrity of free memory and overflow buffers. The range must be specified as no more than two unsigned integers separated by a dash, followed by an optional forward slash and an unsigned integer specifying an event checking frequency. If numbers on either the left side or the right side of the dash are omitted then they will be assumed to be *0* and *infinity* respectively. If the event checking frequency is omitted then it is assumed to be *1*. A value of *0* on its own indicates that no such checking will ever be performed. This option can be used to speed up the execution speed of the library at the expense of checking. Default value: CHECK=*0*.

CHECKALL
: Equivalent to the CHECKALLOCS, CHECKREALLOCS, CHECKFREES and CHECKMEMORY options specified together.

CHECKALLOCS
: Checks that no attempt is made to allocate a block of memory of size zero. A warning will be issued for every such case.

CHECKFORK
: Checks at every call to see if the process has been forked in case new log, profiling and tracing output files need to be started. This option only has an effect on UNIX platforms, but should not be used in multithreaded programs if each thread has a different process identifier.

CHECKFREES
: Checks that no attempt is made to deallocate a NULL pointer. A warning will be issued for every such case.

CHECKMEMORY
: Checks that no attempt is made to perform a zero-length memory operation on a NULL pointer.

CHECKREALLOCS
: Checks that no attempt is made to reallocate a NULL pointer or resize an existing block of memory to size zero. Warnings will be issued for every such case.

DEFALIGN=*unsigned integer*
: Specifies the default alignment for general-purpose memory allocations, which must be a power of two (and will be rounded up to the nearest power of two if it is not). The default alignment for a particular system is calculated at run-time.

EDIT Specifies that a text editor should be invoked to edit any relevant source files that are associated with any warnings or errors when they occur. Only diagnostics which occur at source lines in the program will be affected and only then if they contain source-level information. This option is currently only available on UNIX platforms as it makes use of the mpedit command. It also overrides the behaviour of the LIST option and affects the behaviour of the __mp_view function.

FAILFREQ=*unsigned integer*
: Specifies the frequency at which all memory allocations will randomly fail. For example, a value of *10* will mean that roughly 1 in 10 memory allocations will fail, but a value of *0* will disable all random failures. This option can be useful for stress-testing an application. Default value: FAILFREQ=*0*.

FAILSEED=*unsigned integer*
: Specifies the random number seed which will be used when determining which memory allocations will randomly fail. A value of *0* will instruct the library to pick a random seed every time it is run. Any other value will mean that the random failures will be the same every time the program is run, but only as long as the seed stays the same. Default value: FAILSEED=*0*.

FREEBYTE=*unsigned integer*
: Specifies an 8-bit byte pattern with which to prefill newly-freed memory. This can be used to detect the use of memory which has just been freed. It is also used internally to ensure that freed memory has not been overwritten. Note that the freed memory may be reused the next time a block of memory is allocated and so once memory has been freed its contents are not guaranteed to remain the same as the specified byte pattern. Default value: FREEBYTE=*0x55*.

FREESTOP=*unsigned integer*
: Specifies an allocation index at which to stop the program when it is being freed. When the memory allocation with the specified allocation index is to be freed the program will be halted, and its state may be examined at that point using a suitable debugger. Note that this setting will be ignored if its value is zero. Default value: FREESTOP=*0*.

HELP Displays a quick-reference option summary to the stderr file stream.

LARGEBOUND=*unsigned integer*
: Specifies the limit in bytes up to which memory allocations should be classified as large allocations for profiling purposes. This limit must be greater than the small and medium bounds. Default value: LARGEBOUND=*2048*.

LEAKTABLE
: Specifies that the leak table should be automatically used and a leak table summary should be displayed at the end of program execution. The summary shows a flat profile of all unfreed memory allocations since the start of the program, or since the last call to __mp_clearleaktable if that function was called.

LIMIT=*unsigned integer*
: Specifies the limit in bytes at which all memory allocations should fail if the total allocated memory should increase beyond this. This can be used to stress-test software to see how it behaves in low memory conditions. The internal memory used by the library itself will not be counted as part of the total heap size, but on some systems there may be a small amount of memory required to initialise the library itself. Note that this setting will be ignored if its value is zero. Default value: LIMIT=*0*.

LIST
: Specifies that a context listing should be shown for any relevant source files that are associated with any warnings or errors when they occur. Only diagnostics which occur at source lines in the program will be affected and only then if they contain source-level information. This option is currently only available on UNIX platforms as it makes use of the mpedit command. It also overrides the behaviour of the EDIT option and affects the behaviour of the __mp_view function.

LOGALL
: Equivalent to the LOGALLOCS, LOGREALLOCS, LOGFREES and LOGMEMORY options specified together.

LOGALLOCS
: Specifies that all memory allocations are to be logged and sent to the log file. Note that any memory allocations made internally by the library will not be logged.

LOGFILE=*string*
: Specifies an alternative file in which to place all diagnostics from the mpatrol library. If the LOGDIR environment variable is set and the specified file does not contain a path component in its filename then the log file will be located in the directory specified in LOGDIR. A filename of *stderr* will send all diagnostics to the stderr file stream and a filename of *stdout* will do the equivalent with the stdout file stream. Note that if a problem occurs while opening the log file or if any diagnostics require to be displayed before the log file has had a chance to be opened then they will be sent to the stderr file stream. Default value: LOGFILE=*mpatrol.log* or LOGFILE=*%n.%p.log* if the LOGDIR environment variable is set.

LOGFREES
: Specifies that all memory deallocations are to be logged and sent to the log file. Note that any memory deallocations made internally by the library will not be logged.

LOGMEMORY
: Specifies that all memory operations are to be logged and sent to the log file. These operations will be made by calls to functions such as memset and memcpy. Note that any memory operations made internally by the library will not be logged.

LOGREALLOCS
: Specifies that all memory reallocations are to be logged and sent to the log file. Note that any memory reallocations made internally by the library will not be logged.

MEDIUMBOUND=*unsigned integer*
: Specifies the limit in bytes up to which memory allocations should be classified as medium allocations for profiling purposes. This limit must be greater than the small bound but less than the large bound. Default value: MEDIUMBOUND=*256*.

NOFREE=*unsigned integer*

Specifies that a number of recently-freed memory allocations should be prevented from being returned to the free memory pool. Such freed memory allocations will then be flagged as freed and can be used by the library to provide better diagnostics. If the size of the freed queue is specified as zero then all freed memory will be immediately reused by the mpatrol library. Note that if this option is given a non-zero value then the mpatrol library will always force a memory reallocation to return a pointer to newly-allocated memory, but the expand function will never be affected by this option. Default value: NOFREE=*0*.

NOPROTECT

Specifies that the mpatrol library's internal data structures should not be made read-only after every memory allocation, reallocation or deallocation. This may significantly speed up execution but this will be at the expense of less safety if the program accidentally overwrites some of the library's internal data structures. Note that this option has no effect on systems that do not support memory protection.

OFLOWBYTE=*unsigned integer*

Specifies an 8-bit byte pattern with which to fill the overflow buffers of all memory allocations. This is used internally to ensure that nothing has been written beyond the beginning or the end of a block of allocated memory. Note that this setting will only have an effect if the OFLOWSIZE option is in use. Default value: OFLOWBYTE=*0xAA*.

OFLOWSIZE=*unsigned integer*

Specifies the size in bytes to use for all overflow buffers, which must be a power of two (and will be rounded up to the nearest power of two if it is not). This is used internally to ensure that nothing has been written beyond the beginning or the end of a block of allocated memory. Note that this setting specifies the size for only one of the overflow buffers given to each memory allocation; the other overflow buffer will have an identical size. No overflow buffers will be used if this setting is zero. Default value: OFLOWSIZE=*0*.

OFLOWWATCH

Specifies that watch point areas should be used for overflow buffers rather than filling with the overflow byte. This can significantly reduce the speed of program execution. Note that this option has no effect on systems that do not support watch point areas.

PAGEALLOC=*LOWER|UPPER*

Specifies that each individual memory allocation should occupy at least one page of virtual memory and should be placed at the lowest or highest point within these pages. This allows the library to place an overflow buffer of one page on either side of every memory allocation and write-protect these pages as well as all free and freed memory. Note that this option has no effect on systems that do not support memory protection, and is disabled by default on other systems as it can slow down the speed of program execution.

PRESERVE

Specifies that any reallocated or freed memory allocations should preserve their original contents. This option must be used with the NOFREE option and has no effect otherwise.

PROF Specifies that all memory allocations and deallocations are to be profiled and sent to the profiling output file. Memory reallocations are treated as a memory deallocation immediately followed by a memory allocation.

PROFFILE=*string*

Specifies an alternative file in which to place all memory allocation profiling information from the mpatrol library. If the PROFDIR environment variable is set and the specified file does not contain a path component in its filename then the profiling output file will be located in the directory specified in PROFDIR. A filename of *stderr* will send this information to the stderr file stream and a filename of *stdout* will do the equivalent with the stdout file stream. Note that if a problem occurs while opening the profiling output file then the profiling information will not be output. Default value: PROFFILE=*mpatrol.out* or PROFFILE=*%n.%p.out* if the PROFDIR

environment variable is set.

PROGFILE=*string*
: Specifies an alternative filename with which to locate the executable file containing the program's symbols. On most systems, the library will automatically be able to determine this filename, but on a few systems this option may have to be used before any or all symbols can be read.

REALLOCSTOP=*unsigned integer*
: Specifies a reallocation index at which to stop the program when a memory allocation is being reallocated. If the ALLOCSTOP option is non-zero then the program will be halted when the allocation matching that allocation index is reallocated the specified number of times. Otherwise the program will be halted the first time any allocation is reallocated the specified number of times. Note that this setting will be ignored if its value is zero. Default value: REALLOCSTOP=*0*.

SAFESIGNALS
: Instructs the library to save and replace certain signal handlers during the execution of library code and to restore them afterwards. This was the default behaviour in version 1.0 of the mpatrol library and was changed since some memory-intensive programs became very hard to interrupt using the keyboard, thus giving the impression that the program or system had hung.

SHOWALL
: Equivalent to the SHOWFREE, SHOWFREED, SHOWUNFREED, SHOWMAP and SHOWSYMBOLS options specified together.

SHOWFREE
: Specifies that a summary of all of the free memory blocks should be displayed at the end of program execution. This step will not be performed if an abnormal termination occurs or if there were no free memory blocks.

SHOWFREED
: Specifies that a summary of all of the freed memory allocations should be displayed at the end of program execution. This option must be used in conjunction with the NOFREE option and this step will not be performed if an abnormal termination occurs or if there were no freed allocations.

SHOWMAP
: Specifies that a memory map of the entire heap should be displayed at the end of program execution. This step will not be performed if an abnormal termination occurs or if the heap is empty.

SHOWSYMBOLS
: Specifies that a summary of all of the function symbols read from the program's executable file should be displayed at the end of program execution. This step will not be performed if an abnormal termination occurs or if no symbols could be read from the executable file.

SHOWUNFREED
: Specifies that a summary of all of the unfreed memory allocations should be displayed at the end of program execution. This step will not be performed if an abnormal termination occurs or if there are no unfreed allocations. Note that any marked memory allocations will not be listed.

SMALLBOUND=*unsigned integer*
: Specifies the limit in bytes up to which memory allocations should be classified as small allocations for profiling purposes. This limit must be greater than zero but less than the medium and large bounds. Default value: SMALLBOUND=*32*.

TRACE
: Specifies that all memory allocations, reallocations and deallocations are to be traced and sent to the tracing output file.

TRACEFILE=*string*
: Specifies an alternative file in which to place all memory allocation tracing information from the mpatrol library. If the TRACEDIR environment variable is set and the specified file does not contain a path component in its filename then the tracing output file will be located in the directory

specified in TRACEDIR. A filename of *stderr* will send this information to the stderr file stream and a filename of *stdout* will do the equivalent with the stdout file stream. Note that if a problem occurs while opening the tracing output file then the tracing information will not be output. Default value: TRACEFILE=*mpatrol.trace* or TRACEFILE=*%n.%p.trace* if the TRACEDIR environment variable is set.

UNFREEDABORT=*unsigned integer*

Specifies the minimum number of unfreed allocations at which to abort the program just before program termination. A summary of all the allocations will be displayed on the standard error file stream before aborting. This option may be handy for use in batch tests as it can force tests to fail if they do not free up a minimum number of memory allocations, although marked allocations will not be considered as unfreed allocations. Note that this setting will be ignored if its value is zero. Default value: UNFREEDABORT=*0*.

USEDEBUG

Specifies that any debugging information in the executable file should be used to obtain additional source-level information. This option will only have an effect if the executable file contains a compiler-generated line number table and will be ignored if the mpatrol library was built to support an object file access library that cannot read line tables from object files.

USEMMAP

Specifies that the library should use mmap instead of sbrk to allocate user memory on UNIX platforms. This option should be used if there are problems when using the mpatrol library in combination with another malloc library which uses sbrk to allocate its memory. Memory internal to the mpatrol library is allocated with mmap on systems where it is supported in order to segregate it from user memory, and this behaviour is reversed with the USEMMAP option. It is ignored on systems that do not support the mmap system call.

SEE ALSO

mpatrol(1), mprof(1), mptrace(1), mleak(1), mpsym(1), mpedit(1), hexwords(1), mmap(2), sbrk(2), libmpalloc(3), malloc(3), new(3c++), alloca(3), memory(3), string(3), assert(3), elf(3e), bfd(3).

The mpatrol manual and reference card.

http://www.cbmamiga.demon.co.uk/mpatrol/

AUTHOR

Graeme S. Roy <graeme.roy@analog.com>

COPYRIGHT