mpatrol

COLLABORATORS

	TITLE : mpatrol		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		July 10, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

mpatrol

1.1 mpatrol.guide

```
                mpatrol
******

                                  _               _
                                 | |             | |
 ___   ___   ____   ____   _| |_  ____   ___    | |
| _ ~ _ \ | __ \ / __ | |_  _| | __| / _ \  | |
| | | | | | | | | | | | | | | |   | |   | | | |  | |
| | | | | | |__| | | |__| |  | |   | |   | |__| | |
|_| |_| |_| | ___/ \____|   |_|   |_|   \___/  |_|
            | |
            |_|
```
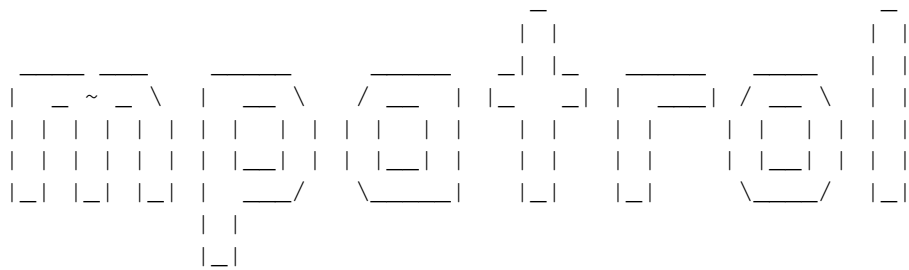
   This document describes mpatrol, a library for controlling and
tracing dynamic memory allocations.

   This is edition 1.9 of the mpatrol manual for version 1.2.0, 16th
May, 2000.

                Foreword
                 Motivation and feedback.

                Overview
                 Overview of mpatrol.

                Features
                 Features of mpatrol.

                Installation
                 Building and installing.

                Integration
                 Debugging existing programs.

                Memory allocations
                 Background.

1.2   mpatrol.guide/Foreword

                    Foreword
****

   I first started writing this library a few years ago when the
company I work for sent me out to a customer who had reported a memory
leak, which he expected was coming from the code generated by our C++
compiler. A few years on and the library has changed dramatically from
its first beginnings, but I thought I'd release it publicly in case
anyone else found it useful.

   When writing the library, I placed more emphasis on the quantity and
quality of information about allocated memory rather than the speed and
efficiency of allocating the actual memory. This means that the
library will use dramatically more memory than normal dynamic memory
allocation libraries and can slow down to a crawl depending on which
options you use. However, the end results are likely to be accurate
and reliable, and in most cases the library will run quite happily at a
sane speed.

   The mpatrol library is by no means the only library of its kind.
Solaris 7 has no less that 6 different malloc libraries, and there are
plenty available as freeware or as commercial products. Try to keep in
mind that mpatrol comes with absolutely no warranty and so if it
doesn't work for you and you need a fast solution, try some of the
other libraries or products available. I have listed some of the most
popular at the end of this manual (see
                    Related software
).

   This manual is arranged so that complete reference material on the
mpatrol library can be found in the appendices, while introductory and
background material can be found in the preceding chapters and
sections. For readers who wish to delve right in and use the library,
the Installation (see
                    Installation
) and Examples (see
                    Examples
)
chapters should be enough to get started in combination with the quick
reference card. Otherwise, this manual should be read from beginning
to end in order to get the most out of the software it describes.

   Due to their very nature, problems with dynamic memory allocations
are notoriously difficult to reproduce and debug, and this is likely to
be the case if you find a bug in the mpatrol library as it might be
extremely hard to reproduce on another system. Details on how to
report bugs are given elsewhere in this document (see
                    Notes
                    ), but it
would be very useful if you could try to provide as much information as
possible when reporting a problem, and that includes having a look in
the library source code to see if it's obvious what is wrong. However,
please try to read the FAQ first in case your question or problem is
covered there since it is usually updated every time I receive a

question about mpatrol.

   The latest version of the mpatrol library and this manual can always
be found at 'http://www.cbmamiga.demon.co.uk/mpatrol/', and any
correspondence relating to mpatrol (bug reports, enhancement requests,
compliments, etc.) should be sent to <mpatrol@cbmamiga.demon.co.uk>.
The mpatrol library is also registered at FreshMeat
('http://freshmeat.net/') so you can receive notification of updates
there as well. I normally only check my e-mail about once or twice a
week, so don't expect an immediate response. I can also be reached at
<graeme@epc.co.uk> but that is my work e-mail address. There is now
also a discussion group at 'http://www.egroups.com/group/mpatrol/'
where you can post mpatrol-related questions but you must first
subscribe to the group before you can send mail to it.

   Note that this manual is not just intended to instruct readers on
how to use the mpatrol library -- it is also written to give a detailed
look at how malloc libraries work in general and how to improve the
efficiency of existing code which uses them. If this subject interests
you, you may find further useful material at The Memory Management
Reference located at 'http://www.harlequin.com/mm/reference/'. It has
links to many documents and research papers in the field of memory
management, and has a large glossary which lists and explains related
terms. You may also wish to look at A Memory Allocator by Doug Lea for
information on general memory allocation principles. It is located at
'http://gee.cs.oswego.edu/dl/html/malloc.html'.

   Finally, I'd like to thank Stephan Springl (<springl@bfw-online.de>)
for his help on reading debugging information from object files via the
GNU BFD library, and Dave Gibson (<david@epc.co.uk>) for his help on
writing thread-safe code. Calum Wilkie (<calum@epc.co.uk>) also
deserves a mention since the idea for providing stack traces comes from
a similar library he wrote a few years ago.

   Oh, and always remember to do final release builds without the
mpatrol library as the library is much slower than normal malloc
implementations and uses much more memory.

   Happy debugging!

   Graeme Roy, 11th October, 1999.

   Edinburgh, Scotland.

1.3 mpatrol.guide/Overview

Overview
****

   The mpatrol library is yet another link library that attempts to
diagnose run-time errors that are caused by the wrong use of
dynamically allocated memory. If you don't know what the malloc()
function or operator new[] do then this library is probably not for

you. You have to have a certain amount of programming expertise and a
knowledge of how to run a command line compiler and linker before you
should attempt to use this.

   Along with providing a comprehensive and configurable log of all
dynamic memory operations that occurred during the lifetime of a
program, the mpatrol library performs extensive checking to detect any
misuse of dynamically allocated memory. All of this functionality can
be integrated into existing code through the inclusion of a single
header file at compile-time. On UNIX and Windows platforms (and
AmigaOS when using gcc) this may not even be necessary as the mpatrol
library can be linked with existing object files at link-time or, on
some platforms, even dynamically linked with existing programs at
run-time.

   All logging and tracing output from the mpatrol library is sent to a
separate log file in order to keep its diagnostics separate from any
that the program being tested might generate. A wide variety of
library settings can also be changed at run-time via an environment
variable, thus removing the need to recompile or relink in order to
change the library's behaviour.

   A file containing a summary of the memory allocation profiling
statistics for a particular program can be produced by the mpatrol
library. This file can then be read by a profiling tool which will
display a set of tables based upon the accumulated data. The profiling
information includes summaries of all of the memory allocations listed
by size and the function that allocated them and a list of memory leaks
with the call stack of the allocating function.

   The mpatrol library has been designed with the intention of
replacing calls to existing C and C++ memory allocation functions as
seamlessly as possible, but in many cases that may not be possible and
slight code modifications may be required. However, a preprocessor
macro containing the version of the mpatrol library is provided for the
purposes of conditional compilation so that release builds and debug
builds can be easily automated.

1.4 mpatrol.guide/Features

Features
****

   An overall list of features contained in the mpatrol library is
given below. This is not intended to be exhaustive since the best way
to see what the library does is to read the documentation and try it
out.

   * Written for UNIX, AmigaOS, Windows and Netware platforms.

   * Can be built to allocate memory from a fixed-sized static array
     rather than using heap memory from the system.

   * Can be built as archive, shared and/or thread-safe libraries on

systems that support them, or even as one large object file. A
lint library can also be built from the mpatrol library on UNIX
platforms.

* 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.

* Contains 14 replacement C dynamic memory allocation functions:

  malloc()   ANSI   Allocates memory.
  calloc()   ANSI   Allocates zero-filled memory.
  memalign()   UNIX   Allocates memory with a specified alignment.
  valloc()   UNIX   Allocates page-aligned memory.
  pvalloc()   UNIX   Allocates a number of pages.
  strdup()   UNIX   Duplicates a string.
  strndup()   old   Duplicates a string with a maximum length.
  strsave()   old   Duplicates a string.
  strnsave()   old   Duplicates a string with a maximum length.
  realloc()   ANSI   Resizes memory.
  recalloc()   old   Resizes memory allocated by calloc().
  expand()   old   Resizes memory but does not relocate it.
  free()   ANSI   Frees memory.
  cfree()   old   Frees memory allocated by calloc().

* Contains 4 replacement C++ dynamic memory allocation functions:

  operator new   Allocates memory.
  operator new[]   Allocates memory for an array.
  operator delete   Frees memory.
  operator delete[]   Frees memory allocated by operator new[].

* Contains 10 replacement C memory operation functions:

  memset()   ANSI   Fills memory with a specific byte.
  bzero()   UNIX   Fills memory with the zero byte.
  memccpy()   UNIX   Copies memory up to a specific byte.
  memcpy()   ANSI   Copies non-overlapping memory.
  memmove()   ANSI   Copies possibly-overlapping memory.
  bcopy()   UNIX   Copies possibly-overlapping memory.
  memcmp()   ANSI   Compares two blocks of memory.
  bcmp()   UNIX   Compares two blocks of memory.
  memchr()   ANSI   Searches memory for a specific byte.
  memmem()   UNIX   Searches memory for specific bytes.

* All of the above functions can also be defined with an additional
  underscore prepended to their external name in order to catch all
  uses of these functions in the system and third-party libraries.

* Contains support for a user-defined low-memory handler function,
  including a replacement for the C++ function, set_new_handler().

* Contains support for user-defined prologue and epilogue callback
  functions, which get called before and after every memory

allocation, reallocation or deallocation.

* A function is provided to return as much information as possible
  about a given memory allocation, and can be called at any time
  during program execution. A similar function is also provided for
  calling from within a debugger and an example command file is
  provided for use with gdb.

* A function is provided to display library settings and heap usage
  statistics, including peak memory usage. This information is also
  displayed at program termination.

* The library reads any user-controllable options at run-time from
  an environment variable, but this does not have to be set as
  defaults will then be used. This prevents having to recompile
  anything in order to change any library settings. An option
  exists to display a quick-reference summary of all of the
  recognised options to the standard error file stream.

* All diagnostics and logging are sent to a file in the current
  directory, but this can be overridden, including forcing the log
  file to be the standard output or standard error file streams.

* Options exist to log details of every memory allocation,
  reallocation or deallocation when they occur.

* Options exist to halt the program at a specific memory allocation,
  reallocation or deallocation when running the program within a
  debugger. These options have no effect when running the program
  without a debugger.

* An option exists to enable memory allocation profiling, which
  forces a summary of all memory allocation statistics to be written
  to a specified file for later use by a profiling command. The
  profiling file can also be written at a specified frequency.

* A profiling command is provided which reads a profiling output
  file produced by the mpatrol library and displays a set of tables
  based on the accumulated data. The profiling information includes
  summaries of all of the memory allocations listed by size and the
  function that allocated them and a list of memory leaks with the
  call stack of the allocating function.

* On UNIX platforms, the mmap() function can optionally be used to
  allocate memory instead of the sbrk() function, but only if the
  system supports it. This can be useful if the mpatrol library
  clashes with another malloc library that uses sbrk() to allocate
  heap memory.

* On non-UNIX platforms where the mpatrol library overrides malloc()
  without requiring the inclusion of mpatrol.h, versions of the UNIX
  functions brk() and sbrk() are provided for compatibility with
  certain libraries. These should not be called by user code as
  they have only limited functionality.

* All newly-allocated memory that is not allocated by the calloc() or
  recalloc() functions will be pre-filled with a non-zero value in

order to catch out programs that wrongly assume that all newly-allocated memory is zeroed. This value can be modified at run-time.

* Can automatically check to see if there have been any illegal writes to bytes located just before and after every memory allocation through the use of overflow buffers. The size of such overflow buffers and the value to pre-fill them with can be modified at run-time. The checks will be performed before every memory allocation call to ensure that nothing has overwritten the overflow buffers, but a function is also provided to perform additional checks under the programmer's control and an option exists to specify a range in which checks will be performed.

* On systems that support them, watch point areas can be used instead of overflow buffers so that every read and write to memory is checked to ensure that it is not within an overflow buffer.

* Can automatically check to see if there have been any illegal writes to free memory blocks. The value to pre-fill free memory blocks with can be modified at run-time. The check will be performed before every memory allocation call to ensure that nothing has overwritten the free memory block, but a function is also provided to perform additional checks under the programmer's control and an option exists to specify a range in which checks will be performed.

* On systems that support memory protection, every memory allocation can optionally be allocated at least one page of memory. That way, any free memory blocks can be made read and write protected so that nothing can access free memory on the heap. An option is provided to specify whether all memory allocations should be allocated at the start or at the end of such pages, and the bytes left over within the pages become overflow buffers.

* All freed memory allocations can optionally be prevented from being returned to the free memory pool. This is useful for detecting if use is being made of freed memory just after a memory allocation has been freed. The contents of the memory allocation can either be preserved or can be pre-filled with a value in order to detect illegal writes to the freed memory allocation.

* Calls to memory operation functions (such as memset() or memcpy()) have their arguments checked to ensure that they do not pass null pointers or attempt to read or write memory straddling the boundary of a previously allocated memory block, although an option exists to turn such an error into a warning so that the operation can still be performed. Tracing from all such functions can also optionally be written to the log file.

* The internal data structures used by the library are kept separate from the rest of the memory allocations. On systems that support memory protection, all of these internal data structures will be write-protected in order to prevent corruption by the calling program. This feature can be overridden at run-time as it can slow the program down.

* Certain signals can be saved and restored on entry to each library
  function and errno is set to ENOMEM if memory cannot be allocated.

* 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.

* A call stack traceback from any function performing a memory
  allocation is stored if the library supports this feature on the
  system it is being run on. This information can then be displayed
  when information about a specific memory allocation is required.
  Two different call stack traceback implementations are provided.

* Symbol table details from executable files and shared libraries are
  automatically read on systems that support this feature in order
  to make the call stack tracebacks more meaningful. An option also
  exists to display a complete list of the symbols that were read by
  the library at program termination.

* Compiler-generated line number tables from any debugging sections
  that exist in executable files and shared libraries can also be
  used by the mpatrol library in order to provide more meaningful
  information in call stack tracebacks.

* If the library is unable to automatically determine a program's
  executable filename to read symbols from then an option exists to
  specify the full path to the program's executable file.

* An option exists to change the default alignment used for
  general-purpose memory allocations.

* Contains support for a user-defined limit to available memory
  which can be useful for stress-testing a program in simulated low
  memory conditions.

* Contains a feature to randomly fail a specific frequency of memory
  allocations which can be useful for stress-testing error recovery
  code in a program.

* An option exists to display a complete memory map of the heap at
  program termination. A function to do this is also available to
  call at any point during program execution.

* Options exist to display all freed and unfreed memory allocations
  at program termination in order to detect memory leaks. A
  separate program is also provided for locating memory leaks in
  unfinished log files.

* An option exists to abort the program with a failure condition if
  there are more than a specified number of unfreed memory
  allocations at program termination. This could be useful for
  batch testing in order to check that all tests free up most of
  their allocated memory.

* Functions always report if their arguments are illegal in order to
  pinpoint any errors, and options exist to perform rigorous

checking of arguments when allocating, reallocating and freeing
memory. In addition, checking is performed to ensure that memory
allocated by operator new[] is not freed with free() for example.

* The type of function performing a memory allocation is always
  stored along with the allocation, as well as the file and line
  number it was called from. If compiled with gcc, the function
  name will also be stored and the thread identifier will be stored
  if using the thread-safe library.

* The library uses a header file to redefine the memory allocation
  functions as macros in order to obtain more information about
  where they were called from. This is not strictly required on
  UNIX and Windows platforms (and AmigaOS when using gcc), since the
  library automatically redefines the default system memory
  allocation functions. All redefinitions in the header can also be
  disabled by defining the NDEBUG preprocessor macro.

* A command is supplied to run a program that was linked with the
  mpatrol library with any specified options on the command line.
  On some UNIX platforms, an option also exists to override the
  default memory allocation routines for any dynamically-linked
  program that was not previously linked with the mpatrol library.

* The mpatrol library can be built to liaise with Parasoft Inuse, a
  commercial graphical memory usage tool that can display the
  current memory map of a running process. Inuse is supplied with
  Parasoft Insure++.

* A small test suite is provided in order to test basic features.

* User documentation is currently available in TeXinfo format as
  well as UNIX manual pages and a quick reference card.

1.5 mpatrol.guide/Installation

Installation
****

   The mpatrol library was initially developed on an Amiga 4000/040
running AmigaOS 3.1. I then installed RedHat Linux 5.1 on my Amiga and
added support for Linux/m68k. I've tried my best to make it as easy as
possible to build and install mpatrol on any system, but it isn't
likely to run smoothly for everybody. However, there shouldn't be any
major problems if you perform the following steps.

  1. Go into the build directory and then into the appropriate
     subdirectory for your system.

  2. Edit the Makefile in that directory and check that it is using the
     appropriate compiler and build tools. The CC macro specifies the
     compiler, the AR macro specifies the tool used to build the archive
     library and the LD macro specifies the tool to build the shared
     library. The CFLAGS macro specifies compiler options that are

always to be used, the OFLAGS macro specifies optimisation options
for the compiler, the SFLAGS macro specifies options to be passed
to the compiler when building a shared library and the TFLAGS
macro specifies options to be passed to the compiler when building
a thread-safe library. You may also need to change the library
names and library build commands on different systems.

3. Use the make command (or equivalent) to build the mpatrol library
   in archive form. The all target builds all possible combinations
   of the mpatrol library for your system. The clean target removes
   all relevant object files from the current directory, while the
   clobber target also removes all libraries that have been built
   from the current directory. On some UNIX platforms, the lint
   target will build a lint library for the mpatrol library.

4. If the mpatrol library is to be built with support for Parasoft
   Inuse then the MP_INUSE_SUPPORT preprocessor macro must be defined
   in the CFLAGS portion of the Makefile before building. This will
   ensure that Inuse will be notified of every memory allocation,
   reallocation and deallocation, but the Insure++ runtime library
   will also have to be linked in with any program that uses mpatrol.

5. Copy all of the libraries that have been built into your local
   library directory. If there were symbolic links created in the
   build directory then these should be recreated in the local
   library directory rather than simply copying them.

6. Copy the mpatrol, mprof and mleak programs that have been built
   into your local bin directory.

7. Go up two directory levels into the src directory and copy the
   mpatrol.h header file into your local include directory.

8. On UNIX platforms, go up one directory level into the man
   directory and copy the man1 and man3 subdirectories to your local
   man directory. Unfortunately, the location for manual pages
   varies from system to system so you may or may not also be able to
   copy the cat1 and cat3 subdirectories as well. The man*
   subdirectories contain the unformatted manual pages while the cat*
   subdirectories contain the formatted manual pages.

9. Go up one directory level into the doc directory and examine the
   files located there. The mpatrol.texi file contains the TeXinfo
   source for this manual and can be translated into a wide variety
   of documentation formats. The refcard.tex file contains the LaTeX
   source for the quick reference card and can be translated into
   formats suitable for printing onto a single page. There may
   already be translated files in the doc directory, but if not you
   will either have to generate them yourself using an appropriate
   tool or you could download an archive containing the latest
   mpatrol manual and reference card in a variety of documentation
   formats from the mpatrol home page. You can then install or print
   these documents.

   Alternatively, the pkg directory contains files that can be used to
automatically generate a package in a specific format suitable for
installation on a system. Two package formats (PKG and RPM) and two

archive formats are currently supported (generic tape archive and LhA).
The first package format is generally used on UNIX SVR4 systems, while
the second was introduced by Red Hat for use in their Linux
distributions. The generic tape archive can be used as a distribution
for UNIX systems where no package format is supported, but it does not
contain information on how to install the files on the system once they
have been extracted from the distribution. The LhA format is roughly
the same, but is intended for Amiga systems and is used for Aminet
distributions. You should really know what you are doing before you
attempt to build a package, and you should also be aware that some of
the package files may need to be modified before you begin.

1.6   mpatrol.guide/Integration

                    Integration
****

   The following steps should allow you to easily integrate the mpatrol
library into an existing application, although some of them may not be
available to do on many platforms. They are listed in the order of
number of changes required to modify existing code -- the last step
will require a complete recompilation of all your code.

  1. This step is currently only available on IRIX, Linux and Solaris
     platforms and on DG/UX 4.20MU07 or later platforms with the
     LD_PRELOAD feature.

     If your program or application has been dynamically linked with
     the system C library (libc.so) or an alternative malloc shared
     library then you can use the -d option to the mpatrol command to
     override the default definitions of malloc(), etc. at run-time
     without having to relink your program.

     For example, if your program's executable file is called testprog
     and it accepts an option specifying an input file, you can force
     the system's dynamic linker to use mpatrol's versions of malloc(),
     etc. instead of the default versions by typing:

         mpatrol -d ./testprog -i file

     The resulting log file should be called mpatrol.<procid>.log by
     default (where PROCID is the current process id), but if no such
     file exists after running the mpatrol command then it will not be
     possible to force the run-time linking of mpatrol functions to
     your program and you will have to proceed to the next step.

  2. This step is currently only available on UNIX and Windows
     platforms (and AmigaOS when using gcc).

     You should be able to link in the mpatrol library when linking
     your program without having to recompile any of your object files
     or libraries, but this will only be worthwhile on systems where
     stack tracebacks are supported, otherwise you should proceed to
     the next step since there will not be enough information for you

to tell where the calls to dynamic memory allocation functions took
place.

Information on how to link the mpatrol library to an application
is given at the start of the examples (see
            Examples
            ), but you
should note that if your program does not directly call any of the
functions in the mpatrol library then it will not be linked in and
you will not see a log file being generated when you run it. You
can force the linking of the mpatrol library by causing malloc()
to be undefined on the link line, usually through the use of the
-u linker option.

3. All of the following steps will require you to recompile some or
   all of your code so that your code calls dynamic memory allocation
   functions from the mpatrol library rather than the system C
   library.

   For this step, if you have a rough idea of where the function
   calls lie that you would like to trace or test, you need only
   recompile the relevant source files. You should modify these
   source files to include the mpatrol.h header file before any calls
   to dynamic memory allocation or memory operation functions.

   However, you should take particular care to ensure that all calls
   to memory allocation functions in the mpatrol library will be
   matched by calls to memory reallocation or deallocation functions
   in the mpatrol library, since if they are unmatched then the log
   file will either fill up with errors complaining about trying to
   free unknown allocations, or warnings about unfreed memory
   allocations at the end of execution.

4. This step requires you to recompile all of your source files to
   include the mpatrol.h header file. Obviously, this will take the
   longest amount of time to integrate, but need not require you to
   change any source files if the compiler you are using has a
   command line option to include a specific header file before any
   source files.

   For example, gcc comes with a -include option which has this
   feature, so if you had to recompile a source file called test.c
   then the following command would allow you to include mpatrol.h
   without having to modify the source file:

        gcc -include /usr/local/include/mpatrol.h -c test.c

   In all cases, it will be desirable to compile your source files with
compiler-generated debugging information since that may be able to be
used by the USEDEBUG option. In addition, more symbolic information
will be available if the executable files have not had their symbol
tables stripped from them, although mpatrol can also fall back to using
the dynamic symbol table from dynamically linked executable files.

1.7 mpatrol.guide/Memory allocations

```
                    Memory allocations
*******************
```

   In the C and C++ programming languages there are generally three
different types of memory allocation that can be used to hold the
contents of variables. Other programming languages such as Pascal,
BASIC and FORTRAN also support some of these types of allocation,
although their implementations may be slightly different.

```
                  Static memory allocations
                   Fixed location, fixed size.

                  Stack memory allocations
                   Variable location, fixed size.

                 Dynamic memory allocations
                  Variable location, variable size.
```

1.8 mpatrol.guide/Static memory allocations

```
Static memory allocations
=========================
```

   The first type of memory allocation is known as a static memory
allocation, which corresponds to file scope variables and local static
variables. The addresses and sizes of these allocations are fixed at
the time of compilation(1) and so they can be placed in a fixed-sized
data area which then corresponds to a section within the final linked
executable file. Such memory allocations are called static because
they do not vary in location or size during the lifetime of the program.

   There can be many types of data sections within an executable file;
the three most common are normal data, BSS data and read-only data.
BSS data contains variables and arrays which are to be initialised to
zero at run-time and so is treated as a special case, since the actual
contents of the section need not be stored in the executable file.
Read-only data consists of constant variables and arrays whose contents
are guaranteed not to change when a program is being run. For example,
on a typical SVR4 UNIX system the following variable definitions would
result in them being placed in the following sections:

```
    int a;          /* BSS data */
    int b = 1;      /* normal data */
    const int c = 2; /* read-only data */
```

   In C the first example would be considered a tentative declaration,
and if there was no subsequent definition of that variable in the
current translation unit then it would become a common variable in the

resulting object file. When the object file gets linked with other
object files, any common variables with the same name become one
variable, or take their definition from a non-tentative definition of
that variable. In the former case, the variable is placed in the BSS
section. Note that C++ has no support for tentative declarations.

    As all static memory allocations have sizes and address offsets that
are known at compile-time and are explicitly initialised, there is very
little that can go wrong with them. Data can be read or written past
the end of such variables, but that is a common problem with all memory
allocations and is generally easy to locate in that case. On systems
that separate read-only data from normal data, writing to a read-only
variable can be quickly diagnosed at run-time.

    ---------- Footnotes ----------

    (1) Or more accurately, at link time.

1.9   mpatrol.guide/Stack memory allocations

Stack memory allocations
========================

    The second type of memory allocation is known as a stack memory
allocation, which corresponds to non-static local variables and
call-by-value parameter variables. The sizes of these allocations are
fixed at the time of compilation but their addresses will vary
depending on when the function which defines them is called. Their
contents are not immediately initialised, and must be explicitly
initialised by the programmer upon entry to the function or when they
become visible in scope.

    Such memory allocations are placed in a system memory area called the
stack, which is allocated per process(1) and generally grows down in
memory. When a function is called, the state of the calling function
must be preserved so that when the called function returns, the calling
function can resume execution. That state is stored on the stack,
including all local variables and parameters. The compiler generates
code to increase the size of the stack upon entry to a function, and
decrease the size of the stack upon exit from a function, as well as
saving and restoring the values of registers.

    There are a few common problems using stack memory allocations, and
most generally involve uninitialised variables, which a good compiler
can usually diagnose at compile-time. Some compilers also have options
to initialise all local variables with a bit pattern so that
uninitialised stack variables will cause program faults at run-time.
As with static memory allocations, there can be problems with reading
or writing past the end of stack variables, but as their sizes are
fixed these can usually easily be located.

    ---------- Footnotes ----------

    (1) Or per thread on some systems.

1.10 mpatrol.guide/Dynamic memory allocations

                    Dynamic memory allocations
===========================

   The last type of memory allocation is known as a dynamic memory
allocation, which corresponds to memory allocated via malloc() or
operator new[]. The sizes, addresses and contents of such memory vary
at run-time and so can cause a lot of problems when trying to diagnose
a fault in a program. These memory allocations are called dynamic
memory allocations because their location and size can vary throughout
the lifetime of a program.

   Such memory allocations are placed in a system memory area called the
heap, which is allocated per process on some systems, but on others may
be allocated directly from the system in scattered blocks. Unlike
memory allocated on the stack, memory allocated on the heap is not
freed when a function or scope is exited and so must be explicitly
freed by the programmer. The pattern of allocations and deallocations
is not guaranteed to be (and is not really expected to be) linear and
so the functions that allocate memory from the heap must be able to
efficiently reuse freed memory and resize existing allocated memory on
request. In some programming languages there is support for a garbage
collector, which attempts to automatically free memory that has had all
references to it removed, but this has traditionally not been very
popular for programming languages such as C and C++, and has been more
widely used in functional languages like ML(1).

   Because dynamic memory allocations are performed at run-time rather
than compile-time, they are outwith the domain of the compiler and must
be implemented in a run-time package, usually as a set of functions
within a linker library. Such a package manages the heap in such a way
as to abstract its underlying structure from the programmer, providing
a common interface to heap management on different systems. However,
this malloc library must decide whether to implement a fast memory
allocator, a space-conserving memory allocator, or a bit of both. It
must also try to keep its own internal tables to a minimum so as to
conserve memory, but this means that it has very little capability to
diagnose errors if any occur.

   In some compiler implementations there is a builtin function called
alloca(). This is a dynamic memory allocation function that allocates
memory from the stack rather than the heap, and so the memory is
automatically freed when the function that called it returns. This is
a non-standard feature that is not guaranteed to be present in a
compiler, and indeed may not be possible to implement on some systems.
However, some compilers now support variable length arrays which
provide roughly the same functionality.

   As can be seen from the above paragraphs, dynamic memory allocations
are the types of memory allocations that can cause the most problems in
a program since almost nothing about them can be used by the compiler
to give the programmer useful warnings about using uninitialised

variables, using freed memory, running off the end of a
dynamically-allocated array, etc. It is these types of memory
allocation problems that the mpatrol library loves to get its teeth
into!

      ---------- Footnotes ----------

   (1) There is currently at least one garbage collection package
available for C and C++ (see
                Related software
).

1.11   mpatrol.guide/Operating system support

                Operating system support
****

   Beneath every malloc library's public interface there is the
underlying operating system's memory management interface. This
provides features which can be as simple as giving processes the
ability to allocate a new block of memory for themselves, or it can
offer advanced features such as protecting areas of memory from being
read or written. Some embedded systems have no operating systems and
hence no support for dynamic memory allocation, and so the malloc
library must instead allocate blocks of memory from a fixed-sized array.
The mpatrol library can be built to support all of the above types of
system, but the more features an operating system can provide it with,
the more it can do.

   On operating systems such as UNIX and Windows, all dynamic memory
allocation requests from a process are dealt with by using a feature
called virtual memory. This means that a process cannot perform
illegal requests without them being denied, which protects the other
running processes and the operating system from being affected by such
errors. However, on AmigaOS and Netware platforms there is no virtual
memory support and so all processes effectively share the same address
space as the operating system and any other running processes. This
means that one process can accidentally write into the data structures
of another process, usually causing the other process to fail and bring
down the system. In addition, a process which allocates a lot of memory
will result in there being less available memory for other running
processes, and in extreme cases the operating system itself.

                Virtual memory
                 Addition facilities.

                Call stacks and symbol tables
                 Tracing information.

                Threads
                 Multiple threads of execution.

1.12   mpatrol.guide/Virtual memory

```
Virtual memory
==============
```

   Virtual memory is an operating system feature that was originally
used to provide large usable address spaces for every process on
machines that had very little physical memory. It is used by an
operating system to fool(1) a running process into believing that it can
allocate a vast amount of memory for its own purposes, although whether
it is allowed to or not depends on the operating system and the
permissions of the individual user.

   Virtual memory works by translating a virtual address (which the
process uses) into a physical address (which the operating system
uses). It is generally implemented via a piece of hardware called a
memory management unit, or MMU. The MMU's primary job is to translate
any virtual addresses that are referred to by machine instructions into
physical addresses by looking up a table which is built by the
operating system. This table contains mappings to and from pages(2)
rather than bytes since it would otherwise be very inefficient to
handle mappings between individual bytes. As a result, every virtual
memory operation operates on pages, which are indivisible and are
always aligned to the system page size.

   Even though each process can now see a huge address space, what
happens when it attempts to allocate more pages than actually
physically exist, or allocate an additional page of memory when all of
the physical pages are in use by it and other processes? This problem
is solved by the operating system temporarily saving one or more of the
least-used pages (which might not necessarily belong that that process)
to a special place in the file system called a swap file, and mapping
the new pages to the physical addresses where the old pages once
resided. The old pages which have been swapped out are no longer
currently accessible, but their location in the swap file is noted in
the translation table.

   However, if one of the pages that has been swapped out is accessed
again, a page fault occurs at the instruction which referred to the
address and the operating system catches this and reloads the page from
the swap file, possibly having to swap out another page to make space
for the new one. If this occurs too often then the operating system
can slow down, having to constantly swap in and swap out the same pages
over and over again. Such a problem is called thrashing and can only
really be overcome by using less virtual memory or buying more physical
memory.

   It is also possible to take advantage of the virtual memory system's
interaction between physical memory and the file system in program
code, since mapping an existing file to memory means that the usual
file I/O operations can be replaced with memory read and write
operations. The operating system will work out the optimum way to read

and write any buffers and it means that only one copy of the file
exists in both physical memory and the file system. Note that this is
how shared libraries(3) on UNIX platforms are generally implemented,
with each individual process that uses the shared library having it
mapped to somewhere in its address space.

   Another major feature of virtual memory is its ability to read
protect and write protect individual pages of process memory. This
means that the operating system can control access to different parts
of the address space for each process, and also means that a process
can read and/or write protect an area of memory when it wants to ensure
that it won't ever read or write to it again. If an illegal memory
access is detected then a signal will be sent to the process, which can
either be caught and handled or will otherwise terminate the process.
Note that as with all virtual memory operations, this ability to
protect memory only applies to pages, so that it is not possible to
protect individual bytes.

   However, some versions of UNIX have programmable software watch
points which are implemented at operating system level. These are
normally used by debuggers to watch a specified area of memory that is
expected to be read from or written to, but can just as easily be used
to implement memory protection at byte level. Unfortunately, as this
feature is implemented in software(4) rather than in hardware, watch
points tend to be incredibly slow, mainly as a result of the operating
system having to check every instruction before it is executed.

   There is also an additional problem when using watch points, which
is due to misaligned reads from memory. These can occur with
compiler-generated code or with optimised library routines where memory
read, move or write operations have been optimised to work at word
level rather than byte level. For example, the memcpy() function would
normally be written to copy memory a byte at a time, but on some
systems this can be improved by copying a word at a time.
Unfortunately, care has to be taken when reading and writing such words
as the equivalent bytes may not be aligned on word boundaries.
Technically, reading additional bytes before or after a memory
allocation when they share the same word is legal, but when using watch
points such errors will be picked up. The mpatrol library replaces
most of the memory operation functions provided by the system libraries
with safer versions, although they may not be as efficient.

   An operating system with virtual memory is usually going to run ever
so slightly slower than an operating system without it(5), but the
advantages of virtual memory far outweigh the disadvantages, especially
when used for debugging purposes.

   ---------- Footnotes ----------

   (1) Well, perhaps that's too harsh a word, but it will certainly seem
that way to a process running on a 32-bit UNIX system with only 4
megabytes of physical memory, and yet it will be able to read from and
write to over 4 gigabytes of virtual memory!

   (2) The size of a page varies between operating systems and
processor architectures, but they are generally around 4 or 8 kilobytes
in size, and are always a power of two.

   (3) DLLs on Windows platforms.

   (4) The operating system is still considered software.

   (5) Due to the overhead of having to translate every address and
swap in and out pages -- although memory mapped files will usually be
more efficient than using normal file operations on a system without
virtual memory.

1.13   mpatrol.guide/Call stacks and symbol tables

                  Call stacks and symbol tables
============================

   As stated in the section on stack memory allocations (see

                  Stack memory allocations
                  ), when a function is called, a copy of the
caller's state information (including local variables and registers) is
saved on the stack so that it can be restored when the called function
returns. On many operating systems there is a calling convention(1)
which defines the layout of such stack entries so that code compiled in
different languages and with different compilers can be intermixed.
This usually specifies at which stack offsets the stack pointer,
program counter and local variables for the calling function can be
found, although on some processor architectures the function calling
conventions are specified by the hardware and so the operating system
must use these instead.

   On systems that have consistent calling conventions, it is usually
possible to perform call stack tracebacks from within the current
function in order to determine the stack of function calls that led to
the current function. This is extremely useful for debugging purposes
and is done by examining the current stack frame to see if there is a
pointer to the previous stack frame. If there is, then it can be
followed to find out all of the state information about the calling
function. This can be repeated until there are no more stack frames.
This is generally how this information is determined by debuggers when
a call stack traceback is requested.

   In addition to the pointer to the previous stack frame, the saved
state information also always contains the saved program counter
register, which contains either the address of the instruction that
performed the function call, or the address of the instruction at which
to continue execution when the called function returns(2). This
information can be used to identify which function performed the call,
since the address of the instruction must lie between the start and end
of one of the functions in the process.

   However, in order to determine this symbolic information, it must be
possible to find out where the start and end addresses of all of the
functions in the process are. This can usually only be read from
object files, since they contain the symbol tables that were used by

the linker to generate the final executable file for the program. The
object file's symbol tables normally contain information about the
start address, size, name and visibility of every symbol that was
defined, but this depends on the format of the object file and if the
symbol tables have been stripped from the final executable file.

   If the object file was created by a compiler then it may also
contain debugging information that was generated by the compiler for
use with a debugger. Such information may include a mapping of code
addresses to source lines(3), and this information can be used by the
mpatrol library to provide more meaningful information in call stack
tracebacks.

   On systems that support shared libraries, additional work must be
done to determine the symbolic information for all of the functions
which have been defined in them. The symbols for functions that are
defined in shared libraries normally appear as undefined symbols in the
executable file for the program and so must be searched in the system
in order to get the necessary information. It is usually necessary to
liaise with the dynamic linker(4) on many systems.

   ---------- Footnotes ----------

   (1) Usually part of the Application Binary Interface, or ABI.

   (2) Also known as the return address.

   (3) Generally known as a line number table.

   (4) Which is the part of the operating system that performs the
run-time linking of shared libraries.

1.14 mpatrol.guide/Threads

Threads
=======

   On systems with virtual memory, such as UNIX and Windows, user
programs are run as processes which have their own address space and
resources. If a process needs to create sub-processes to perform other
tasks it must call fork() or spawn() to create new processes, but these
new processes do not share the same address space or resources as the
parent process. If processes need to share memory they must either use
a message passing interface or explicitly mark a range of memory as
shareable.

   Traditionally, this was not too much of a handicap as parallel
processing was an expensive luxury and could only be made use of by the
kernel of such systems. However, with the birth of fast processors and
parallel programming, programs could be made to run more efficiently
and faster on multi-processor systems by having more than one thread of
control. This was achieved by allowing processes to have more than one
program counter through which the processor could execute instructions,

and if one thread of control stalled for a particular reason then
another could continue without stalling the entire process.

   Such multithreaded programs allow parallel programming and implicit
shared memory between threads since all threads in a process share the
same address space and resources. This is similar to operating systems
that have no virtual memory, such as AmigaOS and Netware(1), except
that once a process terminates, all threads terminate as well and all
of its resources are still reclaimed.

   Multithreaded programming generally needs no compiler support, but
does require some primitive operations to be supported by the operating
system for a threads library to call. The functions that are available
in the threads library provide the means for a process to create and
destroy threads. There are currently several popular threads libraries
available, although the POSIX threads standard remains the definitive
implementation.

   It is always important to remember when programming a multithreaded
application that because all threads in a process share the same
address space, measures must be taken to prevent threads reading and
writing global data in a haphazard fashion. This can either be done by
locking with semaphores and mutexes, or can be performed by using stack
variables instead of global variables since every thread has its own
local stack. Care must be taken to write re-entrant functions -- i.e.
a function will give exactly the same result with one thread as it will
with multiple threads running it at the same time.

       ---------- Footnotes ----------

   (1) Where the kernel is effectively a single process running all
user programs as threads.

1.15 mpatrol.guide/Using mpatrol

                Using mpatrol
****

   This chapter contains a general description of all of the features
of mpatrol and how to use them effectively. You'll also find a
complete reference for mpatrol in the appendices, but you may wish to
try out the examples (see
                Examples
) and the tutorial (see
                Tutorial
)
before reading further.

                Library behaviour
                 Miscellaneous information.

                Logging and tracing

1.16   mpatrol.guide/Library behaviour

```
Library behaviour
=================
```

Most of the behaviour of the mpatrol library can be controlled at
run-time via options which are read from the MPATROL_OPTIONS
environment variable. This prevents you having to recompile or relink
each time you want to change a library setting, and so makes it really
easy to try out different settings to locate a particular bug. You
should know how to set the value of an environment variable on your
system before you read on.

By default, the mpatrol library will attempt to determine the
minimum required alignment for any generic memory allocation when it
first initialises itself. This may be affected by the compiler and its
settings when the library was built but it should normally reflect the
minimum alignment required by the processor on your system. If you
would prefer a larger (or perhaps even smaller) default alignment you
may change it at run-time using the DEFALIGN option. The value you
supply must be in bytes, must be a power of two, and should not be
larger that the system page size. If you encounter bus errors due to
misaligned memory accesses then you should increase this value.

On systems that have virtual memory the library will attempt to
write-protect all of its internal structures when user code is being
run. This ensures that it is nearly impossible for a program to
corrupt any mpatrol library data. However, unprotecting and then
protecting the structures at every library call has a slight overhead
so you may prefer to disable this behaviour by using the NOPROTECT
option. This has no effect on systems that have no virtual memory.

Usually it is desirable for many system library routines to be

protected from being interrupted by certain signals since they may
themselves be called from signal handlers. If this is not the case
then it may be possible to interrupt the program from within such
routines, perhaps causing problems if their global variables are left
in an undefined state. As the mpatrol library replaces some of these
system library routines it is also possible to specify that they are
protected from certain interrupt signals using the SAFESIGNALS option.
However, this can sometimes result in it being hard to interrupt the
program from the keyboard if a lot of processor time is spent in
mpatrol routines, which is why this behaviour is disabled by default(1).

    On UNIX systems, the usual way for malloc libraries to allocate
memory from the process heap is through the sbrk() system call. This
allocates memory from a contiguous heap, but has the disadvantage in
that other library functions may also allocate memory using the same
function, thus creating holes in the heap. This is not a problem for
mpatrol, but you may have a suspicion that your bug is due to a
function from another library corrupting your data so you may wish to
use the USEMMAP option. This is only available on systems that have
the mmap() system call and allows mpatrol to allocate all of its memory
from a part of the process heap that is non-contiguous (i.e. each call
to mmap() may return a block of memory that is completely unrelated to
that returned by the previous call).

    By default, every time an mpatrol library function is called the
library will automatically check the freed memory and overflow buffers
of every memory allocation, which can slow program execution down,
especially if you suspect the error you are looking for occurs at the
1000th memory allocation, for example. You can therefore use the CHECK
option to specify a range of memory allocations at which the mpatrol
library will automatically check the freed memory and overflow buffers.
All other allocations that fall outside this range will not be checked.

    If the mpatrol library that was built for your system supports
reading symbolic information from a program's executable file, but it
cannot locate the executable file, or you wish to specify an
alternative, you can use the PROGFILE option to do this. All this does
is instruct the mpatrol library to read symbols from this file instead.
Note that on systems that support dynamic linking, the library can
also read symbols from a dynamically linked executable file that has
had its normal symbol table stripped.

    Finally, a list of all of the recognised options in the mpatrol
library can be displayed to the standard error file stream by using the
HELP option. This will not affect the settings of the library in any
way, so you should be able to use other options at the same time.

    ---------- Footnotes ----------

    (1) In mpatrol release 1.0 it was enabled by default.

1.17   mpatrol.guide/Logging and tracing

                        Logging and tracing
        ===================

    If you would like to see a complete log of all of the memory
allocations, reallocations and deallocations performed by your program,
use the LOGALL option. This provides detailed tracing for each of the
mpatrol library functions, and a full description of the format of such
tracing is given in Example 1 (see
                        Example 1
). Alternatively, you may
select one or more types of functions to be traced using the LOGALLOCS,
LOGREALLOCS, LOGFREES and LOGMEMORY options if you feel that the log
file is too large when LOGALL is used. By default all diagnostics from
the mpatrol library get sent to mpatrol.log in the current directory,
but this can be changed using the LOGFILE option.

    On systems that support it, every log entry also contains a call
stack traceback that may also include the names of the symbols that
appear on the call stack. If the object file access library that
mpatrol was built with has support for reading line number tables from
object files then the USEDEBUG option will also try to determine the
file name and line number for each entry in the call stack, but only if
the object files contain the relevant debugging information. This
information will only be available before program termination and so
any call stack tracebacks that appear after the library summary will
not be displayed with their corresponding file name and line number.
This option will also slow down program execution since a search
through the line number tables will have to be made every time a call
stack is displayed.

    The mpatrol library will always try to display as much useful
information as possible in this log file, and will always display a
summary of library settings and statistics when your program terminates
successfully. If you don't get this then your program did not call
exit() and either called abort() or was terminated by the operating
system instead. In such cases, either use a debugger to see where your
program crashed or use the LOGALL option to see the last successful
library call in the log file so that you have a rough idea of where
your program crashed.

    It is also possible to get mpatrol to write more summary information
to the log file after it writes out its settings and statistics at
program termination. Use the SHOWFREED and SHOWUNFREED options to
display a list of freed and unfreed memory allocations. The former
will only be displayed if the NOFREE option is used, but the latter can
be useful for detecting memory leaks. The SHOWMAP option will display
a memory map of the heap that was valid when the process terminated,
and the SHOWSYMBOLS option will display any symbolic information that
the mpatrol library managed to obtain from any executable files and
libraries that were relevant to the program being tested. All of these
options can be selected with the SHOWALL option.

1.18   mpatrol.guide/General errors

```
General errors
==============
```

   By default, the mpatrol library follows the guidelines for ANSI C
regarding the behaviour of the dynamic memory allocation functions it
replaces(1). This means that calling malloc() with a size of zero is
allowed, for example. However, warnings can be generated for all of
these types of calls by using the CHECKALL option. The CHECKALLOCS
option warns only about calls to malloc() and similar functions with a
size of zero, the CHECKREALLOCS option warns only about calls to
realloc() and similar functions with either a null pointer or a size of
zero, and the CHECKFREES option warns only about calls to free() and
similar functions with a null pointer.

   All newly-allocated memory can be pre-filled with a specified byte
by using the ALLOCBYTE option. This can be used to catch out code that
expects newly-allocated memory to be zeroed, although this option will
have no effect on memory that was allocated with calloc(). All free
memory can also be pre-filled with a different specified byte by using
the FREEBYTE option. This will catch out code that expects to be able
to use the contents of freed memory.

   Alternatively, the mpatrol library can be instructed to keep all
freed memory allocations so that its diagnostics can be clearer about
which freed allocation a piece of code is erroneously trying to access.
This is controlled with the NOFREE option, but since it never reuses
any freed allocations it can result in a lot more heap memory being
used. Note that this option distinguishes between free memory and
freed memory. Free memory is unallocated memory that has been taken
from the system heap. Freed memory is a freed memory allocation, with
all of the original details of the allocation preserved.

   Normally, the NOFREE option will fill the freed allocation with the
free byte so that any code that accesses it will hopefully fall over.
However, the original contents can be preserved using the PRESERVE
option in case you need to see what the contents were just before it
was freed. The NOFREE option is also affected by the PAGEALLOC option,
since then the freed allocation will have its contents both read and
write protected so that nothing can access them. If the PRESERVE
option is used in this case then the freed allocation will only be made
write-protected so that the original contents can be read from but not
written to.

       ---------- Footnotes ----------

   (1) I attempted to do the same for ANSI C++ but there are still
namespace and exception handling issues to be resolved.

1.19   mpatrol.guide/Overwrites and underwrites

```
                    Overwrites and underwrites
```

==========================

   Once a block of memory has been allocated, it is imperative that the
program does not attempt to write any data past the end of the block or
write any data just before the beginning of the block. Even writing a
single byte just beyond the end of an allocation or just before the
beginning of an allocation can cause havoc. This is because most
malloc libraries store the details of the allocated block in the first
few words before the beginning of the block, such as its size and a
pointer to the next block. The mpatrol library does not do this, so a
program which failed using the normal malloc library and worked when
the mpatrol library was linked in is a possible candidate for turning
on overflow buffers.

   Such memory corruption can be extremely difficult to pinpoint as it
is unlikely to show itself until the next call is made to the malloc
library, or if the internal malloc library blocks were not overwritten,
the next time the data is read from the block that was overwritten. If
the former is the case then the next library call will cause an
internal error or a crash, but only when the memory block that was
affected is referenced. This is likely to disappear when using the
mpatrol library since it keeps its internal structures separate, and
write-protects them on systems that support memory protection.

   In order to identify such errors, it is possible to place special
buffers(1) on either side of every memory allocation, and these will be
pre-filled with a specified byte. Before every mpatrol library call,
the library will check the integrity of every such overflow buffer in
order to check for a memory underwrite or overwrite. Depending on the
number of allocations and size of these buffers, this can take a
noticable amount of time (which is why overflow buffers are disabled by
default), but can mean that these errors get noticed sooner. The
option which governs this is OFLOWSIZE. The byte with which they get
pre-filled can be changed with OFLOWBYTE. Depending on what gets
written, it might only be possible to see such errors when a different
size of buffer or a different pre-fill byte is used.

   A worse situation can occur when it is only reads from memory that
overflow or underflow; i.e. with the faulty code reading just before or
just past a memory allocation. These cannot be detected by overflow
buffers as it is not possible using conventional means to interrupt
every single read from memory. However, on systems with virtual
memory, it is possible to use the memory protection feature to provide
an alternative to overflow buffers, although at the added expense of
increased memory usage.

   The PAGEALLOC option turns on this feature and automatically rounds
up the size of every memory allocation to a multiple of the system page
size. It also rounds up the size of every overflow buffer to a
multiple of the system page size so that every memory allocation
occupies its own set of pages of virtual memory and no two memory
allocations occupy the same page of virtual memory. The overflow
buffers are then read and write protected so that any memory accesses
to them will generate an error(2). Following on from the previous
section, the PAGEALLOC option also causes free memory to be read and
write protected as well since that will also occupy non-overlapping
virtual memory pages.

The remaining memory that is left over within an allocation's pages
is effectively turned into traditional overflow buffers, being
pre-filled with the overflow byte and checked periodically by the
mpatrol library to ensure that nothing has written into them. However,
because of this remaining memory, the library has a choice of where to
place the memory allocation within its pages. If it places the
allocation at the very beginning then it will catch memory underwrites,
but if it places the allocation at the very end then it will catch
memory overwrites. Such a choice can be controlled at run-time by
supplying an argument to the PAGEALLOC option. If PAGEALLOC=LOWER is
used then every allocation will be placed at the very beginning of its
pages and if PAGEALLOC=UPPER is used then the placement will be at the
very end of its pages. This is probably better explained in Example 3
(see
                Example 3
) where the problems with PAGEALLOC=UPPER and alignment
are also discussed.

Obviously, there are still some deficiencies when using PAGEALLOC
since it can use up a huge amount of memory (especially with NOFREE)
and the overflow buffers within an allocation's pages can still be read
without causing an immediate error. Both of these deficiencies can be
overcome by using the OFLOWWATCH option to install software watch
points instead of overflow buffers, but there are still very few
systems that support software watch points at the moment, and it can
slow a program's execution speed down by a factor of around 10,000.
The reason for this is that software watch points instruct the
operating system to check every read from and write to memory, which
means that it has to single-step through a process checking every
instruction before it is executed. However, this is a very thorough
way of checking for overflows and is unlikely to miss anything,
although there may be problems with misaligned memory accesses when
using watch points (see
                Virtual memory
).

Note that from release 1.1.0 of mpatrol, the library comes with
replacement functions for many memory operation functions, such as
memset() and memcpy(). These new functions provide additional checks
to ensure that if a memory operation is being performed on a memory
block, the operation will not read or write before or beyond the
boundaries of that block.

Normally, if an error is discovered in the call to such functions,
the mpatrol library will report the error but prevent the operation
from being performed before continuing execution. If the error was
that the range of memory being operated on overflowed the boundaries of
an existing memory allocation then the ALLOWOFLOW option can be used to
turn the error into a warning and force the operation to continue.
This behaviour can be desirable in certain cases where third-party
libraries are being used that make such calls but the end result does
not overflow the allocation boundary.

To conclude, if you suspect your program has a piece of code which
is performing illegal memory underwrites or overwrites to a memory
allocation you should use each of the following options in sequence,

but only if your system supports them.

  1. OFLOWSIZE=8

  2. OFLOWSIZE=32

  3. OFLOWSIZE=1 PAGEALLOC=LOWER

  4. OFLOWSIZE=1 PAGEALLOC=UPPER

  5. OFLOWSIZE=8 OFLOWWATCH

  6. OFLOWSIZE=32 OFLOWWATCH

   ---------- Footnotes ----------

  (1) Commonly known as overflow buffers or fence posts.

  (2) This is a feature that was first used by Electric Fence (see

            Related software
) to track down memory corruption.

1.20   mpatrol.guide/Using with a debugger

Using with a debugger
=====================

    If you would like to use mpatrol to pause at a specific memory
allocation, reallocation or deallocation in a debugger then this
section will describe how to go about it. Unfortunately, debuggers
vary widely in function and usage and are normally very
system-dependent. The example below will use gdb as the debugger, but
as long as you know how to set a breakpoint within a debugger, any one
will do.

    First of all, decide where you would like the mpatrol library to
pause when running your program within the debugger. You can choose
one allocation index to break at using the ALLOCSTOP option, or you can
choose to break at a specific reallocation of that allocation by also
using the REALLOCSTOP option. If you use REALLOCSTOP without using
ALLOCSTOP then you will break at the first memory allocation that has
been reallocated the specified number of times. You can also choose to
break at the point in your program that frees a specific allocation
index by using the FREESTOP option.

    The normal process for determining where you would like to pause
your program in the debugger is by using the LOGALL option and
examining the log file produced by mpatrol. If your program crashed
then you should look at the last entry in the log file to see what the
allocation index (and possibly also the reallocation index) of the last
successful call was. You can then decide which of the above options to
use. Note that the debugger will break at a point before any work is

done by the mpatrol library for that allocation index so that you can
see if it was the last successful operation that caused the damage.

    Having decided which combination of mpatrol options to use, you
should set them in the MPATROL_OPTIONS environment variable before
running the debugger on your program. Alternatively, your debugger may
have a command that allows you to modify your environment during
debugging, but you're just as well setting the environment variable
before you run the debugger as it shouldn't make any difference(1).

    After you get to the debugger command prompt, you should set a
breakpoint at the __mp_trap() function. This is the function that gets
called when the specified allocation index and/or reallocation index
appears and so when you run your program under the debugger the mpatrol
library will call __mp_trap() and the debugger will stop at that point.
If you are not running your program within a debugger, or if you
haven't set the breakpoint, then __mp_trap() will still be called, but
it won't do anything. Note that there may be some naming issues on
some platforms where the visible name of a global function gets an
underscore prepended to it. You may have to take that into account
when setting the breakpoint on such systems.

    Now that you have set the MPATROL_OPTIONS environment variable and
have set the debugger to break at __mp_trap(), all that is required is
for you to run your program. Hopefully, the debugger should stop at
__mp_trap(). If it doesn't then you may have to check your environment
variable settings to ensure that they are the same as when you ran the
program outwith the debugger, although obviously with the addition of
ALLOCSTOP, etc. Once the program has been halted by the debugger, you
can then single-step through your code until you see where it goes
wrong. If this is near the end of your program then you'll have saved
yourself a lot of time by using this method.

    The following example will be used to illustrate the steps involved
in using the ALLOCSTOP, REALLOCSTOP and FREESTOP options. However, it
is only for tutorial purposes and the same effect could easily be
achieved by breaking at line 18 in a debugger because in this case it
is obvious from the code and the mpatrol log file where it is going
wrong. In real programs this is hardly ever the case(2).

```
 1  /*
 2   * Allocates 1000 blocks of 16 bytes, freeing each block immediately
 3   * after it is allocated, and freeing the last block twice.
 4   */


 7  #include "mpatrol.h"


10  int main(void)
11  {
12      void *p;
13      int i;

15      for (i = 0; i < 1000; i++)
16          if (p = malloc(16))
17              free(p);
```

```
    18      free(p);
    19      return EXIT_SUCCESS;
    20  }
```

   Compile this example code with debugging information enabled and
link it with the mpatrol library, then set MPATROL_OPTIONS to LOGALL
and run the resulting program. If you examine mpatrol.log you will see
the following near the bottom of the file.

```
    ...

    ALLOC: malloc (1000, 16 bytes, 2 bytes) [main|test.c|16]
            0x80000D8E main
            0x80000D24 _start

    returns 0x80033000

    FREE: free (0x80033000) [main|test.c|17]
            0x80000DBE main
            0x80000D24 _start

        0x80033000 (16 bytes) {malloc:1000:0} [main|test.c|16]
            0x80000D8E main
            0x80000D24 _start

    FREE: free (0x80033000) [main|test.c|18]
            0x80000DE8 main
            0x80000D24 _start

    ERROR: free: 0x80033000 has not been allocated

    ...
```

   In this example, we'll want to use ALLOCSTOP to halt the program at
the 1000th memory allocation so that we can step through it with a
debugger. So, set MPATROL_OPTIONS to ALLOCSTOP=1000 and load the
program into the debugger. If you are using gdb you can now do the
following steps, but if you are not you will have to use the equivalent
commands in your debugger. Note that (gdb) is the debugger command
prompt and so anything that appears on that line after that should be
typed as a command.

```
    (gdb) break __mp_trap
    Breakpoint 1 at 0x80004026
    (gdb) run
    Starting program: a.out
    Breakpoint 1, 0x80004026 in __mp_trap()
    (gdb) backtrace
    #0  0x80004026 in __mp_trap()
    #1  0x800027ec in __mp_getmemory()
    #2  0x80001138 in __mp_alloc()
    #3  0x80000d8e in main() at test.c:16
    (gdb) finish
    Run till exit from #0  0x80004026 in __mp_trap()
    0x800027ec in __mp_getmemory()
    (gdb) finish
    Run till exit from #0  0x800027ec in __mp_getmemory()
```

```
    0x80001138 in __mp_alloc()
    (gdb) finish
    Run till exit from #0   0x80001138 in __mp_alloc()
    0x80000d8e in main() at test.c:16
    16                  if (p = malloc(16))
    (gdb) step
    17                          free(p);
    (gdb) step
    15              for (i = 0; i < 1000; i++)
    (gdb) step
    18              free(p);
    (gdb) quit
    The program is running.  Quit anyway (and kill it)? (y or n) y
```

   After setting the breakpoint and running the program, the debugger
halts at __mp_trap(). Because __mp_trap() is a function within the
mpatrol library, you don't want to bother stepping through any of the
library functions, and in this case you can't since the mpatrol library
was not compiled with debugging information enabled. So, after
returning from all of the library functions, the source line becomes
line 16 because that was the location of the 1000th memory allocation.
Single-stepping twice gets us to line 18 which is our destination.

   Sometimes it is useful to be able to see information about a memory
allocation whilst running a program from within a debugger. The
__mp_printinfo() function is provided for that purpose and takes a heap
address as its only argument. Using the above example, it would have
been possible to print out information about the pointer p at line 17
from within gdb:

```
    (gdb) call __mp_printinfo(p)
    address 0x80033000 located in allocated block:
        start of block:     0x80033000
        size of block:      2 bytes
        allocated by:       malloc
        allocation index:   1000
        reallocation index: 0
        calling function:   main
        called from file:   test.c
        called at line:     16
        function call stack:
            0x80000D8E main
            0x80000D24 _start
```

   Some debuggers, such as gdb, also allow you to define your own
commands for use in a debugging session. The following example defines
a new gdb command called printalloc which calls __mp_printinfo()(3):

```
    (gdb) define printalloc
    Type commands for definition of "printalloc".
    End with a line saying just "end".
    >call __mp_printinfo($arg0)
    >end
    (gdb) document printalloc
    Type documentation for "printalloc".
    End with a line saying just "end".
    >Displays information about an address in the heap.
```

```
    >end

    ---------- Footnotes ----------

    (1) Unless you've linked the debugger with the mpatrol library.

    (2) The other reason that this program is simple is because a proper
example would generally involve crashing the program, but on AmigaOS
and Netware that would also involve crashing the system -- not
something you'd want to do whilst trying this out.

    (3) A sample GDB command file for use with mpatrol can be found in
extra/.gdbinit.
```

1.21 mpatrol.guide/Testing

```
Testing
=======
```

The mpatrol library has several features that make it useful when
testing a program's dynamic memory allocations. These are features
that do not help in fixing an existing bug, but rather help to identify
additional bugs that may be lurking in your code.

It is possible to set a simulated upper limit on the amount of heap
memory available to a process with the LIMIT option, which accepts a
size in bytes, but will be disabled when it is zero. This can be
extremely useful for testing a program under simulated low memory
conditions to see how it handles such errors. Of course, you should
set the heap limit to a value less than the amount of actual available
memory otherwise this option will have no effect. Note that the
mpatrol library may use up a small amount of heap memory when it
initialises itself(1) so the value passed to the LIMIT option may need
to be set slightly higher than you would normally expect.

It is also possible to instruct the mpatrol library to randomly fail
a certain number of memory allocations so that you can further test
error handling code in a program. The frequency at which failures
occur can be controlled with the FAILFREQ option, where a value of zero
means that no failures will occur, but any other value will randomly
cause failures. For example, a value of 10 will cause roughly one in
ten failures and a value of 1 will cause every memory allocation to
fail. The random sequence can be made predictable by using the
FAILSEED option. If this is non-zero then the same program run with
the same failure frequency and same failure seed will fail on exactly
the same memory allocations. If this is zero then the failure seed
will itself be set randomly, but you can see its value when the summary
is displayed at program termination.

When running batch tests(2) it is sometimes useful to be able to
detect if there have been any memory leaks. Such leaks should normally
be distinguished from code which has purposely not freed the memory
that it allocated, so there may be a certain expected number of unfreed
allocations at program termination. It may be that you would like to

highlight any additional unfreed allocations since they may be due to
real memory leaks, so the UNFREEDABORT option can be set to a threshold
number of expected unfreed allocations. If the library detects a
number of unfreed allocations higher than this then it will abort the
program at termination so that it fails. All tests that fail in this
way can then be examined after the test suite finishes.

    ---------- Footnotes ----------

    (1) Actually, it's not really the mpatrol library that uses the
memory but the object file access libraries since they call malloc() to
allocate any memory that they require.

    (2) A set of tests that run without user intervention.

1.22   mpatrol.guide/Library functions

Library functions
=================

    Along with the standard set of C and C++ dynamic memory allocation
functions, the mpatrol library also comes with an additional set of
functions which can be used to provide additional information to your
program, and which can be called at various points in your code for
debugging purposes. You must always include the mpatrol.h header file
in order to use these functions, but you can check for a specific
version of the mpatrol library by checking the MPATROL_VERSION
preprocessor macro.

    It is possible to obtain a great deal of information about an
existing memory allocation using the __mp_info() function. This takes
an address as an argument and fills in any details about its
corresponding memory allocation in a supplied structure. The following
example illustrates this (it can be found in tests/pass/test4.c).

```
23  /*
24   * Demonstrates and tests the facility for obtaining information
25   * about the allocation a specific address belongs to.
26   */


29  #include "mpatrol.h"
30  #include <stdio.h>


33  void display(void *p)
34  {
35      __mp_allocstack *s;
36      __mp_allocinfo d;

38      if (!__mp_info(p, &d))
39      {
40          fprintf(stderr, "nothing known about address 0x%08lX\n", p);
41          return;
```

```
42          }
43          fprintf(stderr, "block:   0x%08lX\n", d.block);
44          fprintf(stderr, "size:    %lu\n", d.size);
45          fprintf(stderr, "type:    %lu\n", d.type);
46          fprintf(stderr, "alloc:   %lu\n", d.alloc);
47          fprintf(stderr, "realloc: %lu\n", d.realloc);
48          fprintf(stderr, "func:    %s\n", d.func ? d.func : "NULL");
49          fprintf(stderr, "file:    %s\n", d.file ? d.file : "NULL");
50          fprintf(stderr, "line:    %lu\n", d.line);
51          for (s = d.stack; s != NULL; s = s->next)
52          {
53              fprintf(stderr, "\t0x%08lX: ", s->addr);
54              fprintf(stderr, "%s\n", s->name ? s->name : "NULL");
55          }
56          fprintf(stderr, "freed:   %d\n", d.freed);
57  }


60  void func2(void)
61  {
62      void *p;

64      if (p = malloc(16))
65      {
66          display(p);
67          free(p);
68      }
69      display(p);
70  }


73  void func1(void)
74  {
75      func2();
76  }


79  int main(void)
80  {
81      func1();
82      return EXIT_SUCCESS;
83  }
```

When this is compiled and run, it should give the following output,
although the pointers are likely to be different.

```
block:   0x8000A068
size:    16
type:    0
alloc:   10
realloc: 0
func:    func2
file:    test4.c
line:    64
         0x80000BEC: func2
         0x80000C3E: func1
         0x80000C48: main
```

```
          0x800009E8: _start
    freed:    0
    nothing known about address 0x8000A068
```

As you can see, anything that the mpatrol library knows about any
memory allocation can be obtained for use in your own code, which can
be very useful if you need to write handlers to keep track of memory
allocations, etc. for debugging purposes. It can also be useful to
have this information when running your program within a debugger, so
you can use the __mp_printinfo() function to display information about
a heap address if your debugger supports calling functions from the
command prompt.

It is also possible for you to be able to intercept calls to
allocate, reallocate and deallocate memory for your own purposes. You
can install prologue and epilogue functions that the mpatrol library
will call before and after every time one of its functions is called.
These can be used for additional tracing or simply to add extra checks
to your code. The following code is an example of this and can be
found in tests/pass/test2.c.

```
    23  /*
    24   * Demonstrates and tests the facility for specifying user-defined
    25   * prologue and epilogue functions.
    26   */


    29  #include "mpatrol.h"
    30  #include <stdio.h>


    33  void prologue(const void *p, size_t l)
    34  {
    35      if (p == (void *) -1)
    36          fprintf(stderr, "allocating %lu bytes\n", l);
    37      else if (l == (size_t) -1)
    38          fprintf(stderr, "freeing allocation 0x%08lX\n", p);
    39      else if (l == (size_t) -2)
    40          fprintf(stderr, "duplicating string '%s'\n", p);
    41      else
    42          fprintf(stderr, "reallocating allocation 0x%08lX to %lu bytes\n", ←
    p, l);
    43  }


    46  void epilogue(const void *p)
    47  {
    48      if (p != (void *) -1)
    49          fprintf(stderr, "allocation returns 0x%08lX\n", p);
    50  }


    53  int main(void)
    54  {
    55      void *p, *q;

    57      __mp_prologue(prologue);
```

```
    58          __mp_epilogue(epilogue);
    59      if (p = malloc(16))
    60          if (q = realloc(p, 32))
    61              free(q);
    62          else
    63              free(p);
    64      if (p = (char *) strdup("test"))
    65          free(p);
    66      return EXIT_SUCCESS;
    67  }
```

   Once again, if you compile and run the above code, you should see
the following output.

```
    allocating 16 bytes
    allocation returns 0x8000A068
    reallocating allocation 0x8000A068 to 32 bytes
    allocation returns 0x8000A068
    freeing allocation 0x8000A068
    duplicating string 'test'
    allocation returns 0x8000A068
    freeing allocation 0x8000A068
```

   Along with being able to install prologue and epilogue functions,
you can also install a low-memory handler with the __mp_nomemory()
function, which will be called by the mpatrol library if it ever runs
out of memory during the call to a memory allocation function. This
gives you the opportunity to use that handler to either free up any
unneeded memory or simply to abort, thus removing the need to check for
failed allocations.

   Finally, there are three functions which affect the mpatrol library
globally. The first, __mp_check(), allows you to force an internal
check of the mpatrol library's data structures at any point during
program execution. The other two functions, __mp_memorymap() and
__mp_summary() allow you to force the generation of a memory map or
library statistics at any point in your program, in much the same way
as they would normally be displayed at the end of program execution.

1.23   mpatrol.guide/Utilities


```
Utilities
=========
```

   A command is provided with the mpatrol distribution which can run
programs that have been linked with the mpatrol library, using a
combination of mpatrol options that can be set via the command line.
All of these options but one map directly onto their equivalent
environment variable settings and exist mainly so that the user does
not have to manually change the MPATROL_OPTIONS environment variable.

   The one option that is the exception to this is the -d option, which
can be used to run a program under the control of the mpatrol library,
even if it wasn't originally linked with the mpatrol library. This can

only be done on systems that support dynamic linking and where the
dynamic linker recognises the LD_PRELOAD or _RLD_LIST environment
variables. Even then, it can only be used when the program that is
being run has been dynamically linked with the system C library, rather
than statically linked.

The reason for all of these limitations is that some SVR4 UNIX
platforms have a special feature in the dynamic linker which can be
told to override the symbols from one shared library using the symbols
from another shared library at run-time. In this case, it involves
replacing the symbols for malloc(), etc., in the system C library with
the mpatrol versions, but only if they were marked as undefined in the
original executable file and would therefore have to have been loaded
from libc.so.

However, if a program qualifies for use with the -d option, it means
that you can trace all of its dynamic memory allocations as well as
running it with any of the mpatrol library's debugging options. This
is mainly a toy feature which allows you to view and manipulate the
dynamic memory allocations of programs that you don't have the source
for, but in theory it could be quite useful if you need to debug a
previously released executable and are unable to recompile or relink it.

Note that the mpatrol command must be set up to use the correct
object file format access libraries that are required for your system
if you wish to use the -d option. If the mpatrol library was built with
FORMAT=FORMAT_ELF32 support then it must be told to preload the ELF
access library (normally libelf.so). If it was built with
FORMAT=FORMAT_BFD support then it must be told to preload the GNU BFD
access libraries (normally libbfd.so and libiberty.so). However, if
these libraries only exist on your system in archive form then you must
build libmpatrol.so with these extra libraries incorporated into it so
that there are no dependencies on them at run-time. However, there may
well be problems if the resulting shared library contains
position-dependent code from the archive libraries you incorporated.
The only way to find out is for you to try it and see.

In order to build a shared version of the mpatrol library with
embedded object file format access libraries, you must first modify the
Makefile you would normally use to build the mpatrol library. At the
lines where the linker is invoked to build the shared library, you must
explicitly add any object file format access libraries that you want to
use at the end of the linker command line. This ensures that all
references to such libraries will be resolved at link time rather than
run time. You must then edit the file src/config.h and remove all of
the libraries that you embedded from the definition of the
MP_PRELOAD_LIBS preprocessor macro. Finally, rebuild the shared version
of the mpatrol library and the mpatrol command and see if your efforts
were worth it.

Another utility program that is provided is called mleak and is
useful for detecting memory leaks in log files produced by the mpatrol
library. This program should be used if the mpatrol library could not
finish writing the log file due to abnormal program termination (which
would prevent the SHOWUNFREED option from working), but note that some
of the unfreed allocations might have been freed if the program had
terminated successfully.

The mleak command scans through an mpatrol log file looking for
lines beginning with ALLOC: and FREE: but ignores lines beginning with
REALLOC:, so only the LOGALLOCS and LOGFREES options are necessary when
running a program linked with the mpatrol library. Note that as a
result of this, no attempt is made to account for resizing of memory
allocations and so the total amount of memory used by the resulting
unfreed allocations may not be entirely accurate.

The mleak command takes one optional argument which must be a valid
mpatrol log filename but if it is omitted then it will use mpatrol.log
as the name of the log file to use. The mleak command makes two passes
over the log file so the file must be randomly-accessible. If the
filename argument is given as – then the standard input file stream
will be used as the log file.

1.24 mpatrol.guide/Profiling

Profiling
****

The mpatrol library has the capability to summarise the information
it accumulated about the behaviour of dynamic memory allocations and
deallocations over the lifetime of any program that it was linked and
run with. This summary shows a rough profile of all memory allocations
that were made, and is hence called profiling. There are several other
different kinds of profiling provided with most compilation tools, but
they generally profile function calls or line numbers in combination
with the time it takes to execute them.

Memory allocation profiling is useful since it allows a programmer
to see which functions directly allocate memory from the heap, with a
view to optimising the memory usage or performance of a program. It
also summarises any unfreed memory allocations that were present at the
end of program execution, some of which could be as a result of memory
leaks. In addition, a summary of the sizes and distribution of all
memory allocations and deallocations is available.

Only allocations and deallocations are recorded, with each
reallocation being treated as a deallocation immediately followed by an
allocation. For full memory allocation profiling support, call stack
traversal must be supported in the mpatrol library and all of the
program's symbols must have been successfully read by the mpatrol
library before the program was run. The library will attempt to
compensate if either of these requirements are not met, but the
displayed tables may contain less meaningful information.

Memory allocation profiling is disabled by default, but can be
enabled using the PROF option. This writes all of the profiling data
to a file called mpatrol.out in the current directory at the end of
program execution, but the name of this file can be changed using the
PROFFILE option. Sometimes it can also be desirable for the mpatrol
library to write out the accumulated profiling information in the
middle of program execution rather than just at the end, even if it is

only partially complete, and this behaviour can be controlled with the
AUTOSAVE option. This can be particularly useful when running the
program from within a debugger, when it is necessary to analyse the
profiling information at a certain point during program execution.

   When profiling memory allocations, it is necessary to distinguish
between small, medium, large and extra large memory allocations that
were made by a function. The boundaries which distinguish between
these allocation sizes can be controlled via the SMALLBOUND,
MEDIUMBOUND and LARGEBOUND options, but they default to 32, 256 and
2048 bytes respectively, which should suffice for most circumstances.

   The mprof command is a tool designed to read a profiling output file
produced by the mpatrol library and display the profiling information
that was obtained. The profiling information includes summaries of all
of the memory allocations listed by size and the function that
allocated them and a list of memory leaks with the call stack of the
allocating function.

   Along with the options listed below, the mprof command takes one
optional argument which must be a valid mpatrol profiling output
filename but if it is omitted then it will use mpatrol.out as the name
of the file to use. If the filename argument is given as – then the
standard input file stream will be used as the profiling output file.

–a

     Specifies that different call sites from within the same function
     are to be differentiated and that the names of all functions
     should be displayed with their call site offset in bytes. This
     affects the direct allocation and memory leak tables.

–c

     Specifies that certain tables should be sorted by the number of
     allocations or deallocations rather than the total number of bytes
     allocated or deallocated. This affects the direct allocation and
     memory leak tables.

–n <DEPTH>
     Specifies the maximum stack depth to use when calculating if one
     call site has the same call stack as another call site. This also
     specifies the maximum number of functions to display in a call
     stack. If DEPTH is 0 then the call stack depth will be unlimited
     in size. The default call stack depth is 1. This affects the
     memory leak table.

–V
     Displays the version number of the mprof command.

   We'll now look at an example of using the mpatrol library to profile
the dynamic memory allocations in a program. However, remember that
this example will only fully work on your machine if the mpatrol
library supports call stack traversal and reading symbols from
executable files on that platform. If that is not the case then only
some of the features will be available.

   The following example program performs some simple calculations and
displays a list of numbers on its standard output file stream, but it

serves to illustrate all of the different features of memory allocation
profiling that mpatrol is capable of. The source for the program can
be found in tests/profile/test1.c.

```
23  /*
24   * Associates an integer value with its negative string equivalent in a
25   * structure, and then allocates 256 such pairs randomly, displays them
26   * then frees them.
27   */


30  #include <stdio.h>
31  #include <stdlib.h>
32  #include <string.h>


35  typedef struct pair
36  {
37      int value;
38      char *string;
39  }
40  pair;


43  pair *new_pair(int n)
44  {
45      static char s[16];
46      pair *p;

48      if ((p = (pair *) malloc(sizeof(pair))) == NULL)
49      {
50          fputs("Out of memory\n", stderr);
51          exit(EXIT_FAILURE);
52      }
53      p->value = n;
54      sprintf(s, "%d", -n);
55      if ((p->string = strdup(s)) == NULL)
56      {
57          fputs("Out of memory\n", stderr);
58          exit(EXIT_FAILURE);
59      }
60      return p;
61  }


64  int main(void)
65  {
66      pair *a[256];
67      int i, n;

69      for (i = 0; i < 256; i++)
70      {
71          n = (int) ((rand() * 256.0) / (RAND_MAX + 1.0)) - 128;
72          a[i] = new_pair(n);
73      }
74      for (i = 0; i < 256; i++)
75          printf("%3d: %4d -> \"%s\"\n", i, a[i]->value, a[i]->string);
```

```
76        for (i = 0; i < 256; i++)
77            free(a[i]);
78        return EXIT_SUCCESS;
79  }
```

   After the above program has been compiled and linked with the
mpatrol library, it should be run with the PROF option set in the
MPATROL_OPTIONS environment variable. Note that mpatrol.h was not
included as it is not necessary for profiling purposes.

   If all went well, a list of numbers should be displayed on the
screen and a file called mpatrol.out should have been produced in the
current directory. This is a binary file containing the total amount
of profiling information that the mpatrol library gathered while the
program was running, but it contains concise numerical data rather than
human-readable data. To make use of this file, the mprof command must
be run. An excerpt from the output produced when running mprof with no
options is shown below.

                              ALLOCATION BINS

                           (number of bins: 1024)

                allocated   unfreed
        ------------------------------- -------------------------------
  size count   %   bytes   % count   %   bytes   %

     2   8   1.56   16   0.54   8   3.12   16   1.70
     3   99 19.34   297   9.94   99 38.67   297 31.60
     4   118 23.05   472 15.80   118 46.09   472 50.21
     5   31   6.05   155   5.19   31 12.11   155 16.49
     8   256 50.00   2048 68.54   0   0.00   0   0.00

 total   512   2988   256   940

                            DIRECT ALLOCATIONS

                   (0 < s <= 32 < m <= 256 < l <= 2048 < x)

           allocated   unfreed
     -------------------------- --------------------------
 bytes   % s m l x bytes   % s m l x count function

  2988 100.00 %%   940 100.00 %%   512 new_pair

  2988   %%   940   %%   512 total

                              MEMORY LEAKS

                        (maximum stack depth: 1)

                    unfreed   allocated
    -- ----------------
        %   bytes   % count   %   bytes count function

 100.00   940 31.46   256 50.00   2988   512 new_pair
```

```
              940   31.46     256   50.00       2988      512  total
```

   The first table shown is the allocation bin table which summarises
the sizes of all objects that were dynamically allocated throughout the
lifetime of the program. In this particular case, counts of all
allocations and deallocations of sizes 1 to 1023 bytes were recorded by
the mpatrol library in their own specific bin and this information was
written to the profiling output file. Allocations and deallocations of
sizes larger than or equal to 1024 bytes are counted as well and the
total number of bytes that they represent are also recorded. This
information can be extremely useful in understanding which sizes of
data structures are allocated most during program execution, and where
changes might be made to make more efficient use of the dynamically
allocated memory.

   As can be seen from the allocation bin table, 8 allocations of 2
bytes, 99 allocations of 3 bytes, 118 allocations of 4 bytes, 31
allocations of 5 bytes and 256 allocations of 8 bytes were made during
the execution of the program. However, all of these memory allocations
except the 8 byte allocations were still not freed by the time the
program terminated, resulting in a total memory leak of 940 bytes.

   The next table shown is the direct allocation table which lists all
of the functions that allocated memory and how much memory they
allocated. The s m l x columns represent small, medium, large and
extra large memory allocations, which in this case are 0 bytes is less
than a small allocation, which is less than or equal to 32 bytes, which
is less than a medium allocation, which is less than or equal to 256
bytes, which is less than a large allocation, which is less than or
equal to 2048 bytes, which is less than an extra large allocation. The
numbers listed under these columns represent a percentage of the
overall total and are listed as %% if the percentage is 100% or as . if
the percentage is less than 1%. Percentages of 0% are not displayed.

   The information displayed in the direct allocation table is useful
for seeing exactly which functions in a program directly perform memory
allocation, and can quickly highlight where optimisations can be made
or where functions might be making unnecessary allocations. In the
example, this table shows us that 2988 bytes were allocated over 512
calls by new_pair() and that 940 bytes were left unfreed at program
termination. All of the allocations that were made by new_pair() were
between 1 and 32 bytes in size.

   We could now choose to sort the direct allocation table by the
number of calls to allocate memory, rather than the number of bytes
allocated, with the -c option to mprof, but that is not relevant in
this example. However, we know that there are two calls to allocate
memory from new_pair(), so we can use the -a option to mprof to show
all call sites within functions rather than just the total for each
function. This option does not affect the allocation bin table so the
new output from mprof with the -a option looks like:

                           DIRECT ALLOCATIONS

                    (0 < s <= 32 < m <= 256 < l <= 2048 < x)

          allocated   unfreed

```
     ------------------------   ------------------------
     bytes       %  s  m  l  x  bytes        %  s  m  l  x  count  function

      2048   68.54  69                 0   0.00                256  new_pair+14
       940   31.46  31               940 100.00  %%            256  new_pair+110

      2988          %%               940          %%           512  total
```

                        MEMORY LEAKS

                 (maximum stack depth: 1)

```
                  unfreed                     allocated
     ----------------------------------------  ----------------
         %      bytes       %   count      %    bytes   count  function

     100.00       940  100.00     256  100.00      940     256  new_pair+110

                  940   31.46     256   50.00     2988     512  total
```

   The names of the functions displayed in the above tables now have a
byte offset appended to them to indicate at what position in the
function a call to allocate memory occurred(1). Now it is possible to
see that the first call to allocate memory from within new_pair() has
had all of its memory freed, but the second call (from strdup()) has
had none of its memory freed.

   This is also visible in the next table, which is the memory leak
table and lists all of the functions that allocated memory but did not
free all of their memory during the lifetime of the program. The
default behaviour of mprof is to show only the function that directly
allocated the memory in the memory leak table, but this can be changed
with the -n option. This accepts an argument specifying the maximum
number of functions to display in one call stack, with zero indicating
that all functions in a call stack should be displayed. This can be
useful for tracing down the functions that were indirectly responsible
for the memory leak. The new memory leak table displayed by mprof with
the -a and -n0 options looks like:

                        MEMORY LEAKS

                 (maximum stack depth: 0)

```
                  unfreed                     allocated
     ----------------------------------------  ----------------
         %      bytes       %   count      %    bytes   count  function

     100.00       940  100.00     256  100.00      940     256  new_pair+110
                                                                main+88
                                                                _start+68

                  940   31.46     256   50.00     2988     512  total
```

   Now that we know where the memory leak is coming from, we can fix it
by freeing the string as well as the structure at line 77. A version
of the above program that does not contain the memory leak can be found
in tests/profile/test2.c.

Much of the functionality of this implementation of memory
allocation profiling is based upon mprof by Benjamin Zorn and Paul
Hilfinger, which was written as a research project and ran on MIPS,
SPARC and VAX machines. However, the profiling output files are
incompatible, the tables displayed have a different format, and the way
they are implemented is entirely different.

---------- Footnotes ----------

(1) If no symbols could be read from the program's executable file,
or if the corresponding symbol could not be determined, then the
function names will be replaced with the code addresses at which the
calls took place.

1.25   mpatrol.guide/Improving performance

                    Improving performance
****

   Because of their need to cover every eventuality, malloc library
implementations are very general and most do their job well when you
consider what is thrown at them. However, your program may not be
performing as well as it should simply because there may be a more
efficient way of dealing with dynamic memory allocations. Indeed,
there may even be a more efficient malloc library available for you to
use.

   If you need to allocate lots of blocks of the same size(1), but you
won't know the number of blocks you'll require until run-time then you
could take the easy approach by simply allocating a new block of memory
for each occurrence. However, this is going to create a lot of
(typically small) memory blocks that the underlying malloc library will
have to keep track of, and even in many good malloc libraries this is
likely to cause memory fragmentation and possibly even result in the
blocks scattered throughout the address space rather than all in the
one place, which is not necessarily a good thing on systems with
virtual memory.

   An alternative approach would be to allocate memory in multiples of
the block size, so that several blocks would be allocated at once.
This would require slightly more work on your part since you would need
to write interface code to return a single block, while possible
allocating space for more blocks if no free blocks were available.
However, this approach has several advantages. The first is that the
malloc library only needs to keep track of a few large allocations
rather than lots of small allocations, so splitting and merging free
blocks is less likely to occur. Secondly, your blocks will be
scattered about less in the address space of the process, which means
that on systems with virtual memory there are less likely to be page
faults if you need to access or traverse all of the blocks you have
created.

   A memory allocation concept that is similar to this is called an

arena. This datatype requires functions which are built on top of the
existing malloc library functions and which associate each memory
allocation with a particular arena. An arena can have as many
allocations added to it as required, but allocations cannot usually be
freed until the whole arena is freed. Note that there are not really
any generic implementations of arenas that are available as everyone
tends to write their own version when they require it, although Digital
UNIX and SGI IRIX systems do come with an arena library called amalloc.

    However, what if you don't plan to free all of the blocks at the
same time? A slight modification to the above design could be to have
a slot table. This would involve allocating chunks of blocks as they
are required, adding each individual block within a chunk to a
singly-linked list of free blocks. Then, as new blocks are required,
the allocator would simply choose the first block on the free list,
otherwise it would allocate memory for a new chunk of blocks and add
them to the free list. Freeing individual blocks would simply involve
returning the block to the free list. If this description isn't clear
enough, have a look in src/slots.h and src/slots.c. This is how the
mpatrol library allocates memory from the system for all of its internal
structures. For variable-sized structures, a slightly different
approach needs to be taken, but for an example of this using strings
see src/strtab.h and src/strtab.c.

    Another optimisation that is possible on UNIX and Windows platforms
is making use of memory-mapped files. This allows you to map a
filesystem object into the address space of your process, thus allowing
you to treat a file as an array of bytes. Because it uses the virtual
memory system to map the file, any changes you make to the mapped
memory will be applied to the file. This is implemented through the
virtual memory system treating the file as a pseudo swap file and will
therefore only use up physical memory when pages are accessed. It also
means that file operations can be replaced by memory read and write
operations, leading to a very fast and efficient way of performing I/O.
Another added bonus of this system means that entire blocks of process
memory can be written to a file for later re-use, just as long as the
file can later be mapped to the same address. This can be a lot faster
than writing to and reading from a specific format of file.

    If you really don't want to keep track of dynamic memory allocations
at all then perhaps you should consider garbage collection. This
allows you to make dynamic memory allocations that need not necessarily
be matched by corresponding calls to free these allocations. A garbage
collector will (at certain points during program execution) attempt to
look for memory allocations that are no longer referenced by the
program and free them for later re-use, hence removing all possibility
of memory leaks. However, the garbage collection process can take a
sizable chunk of processor time depending on how large the program is,
so it is not really an option for real-time programming. It is also
very platform-dependent as it examines very low-level structures within
a process in order to determine which pointers point to which memory
allocations. But there is at least one garbage collector(2) that works
well with C and C++ and acts as a replacement for malloc() and free(),
so it may be the ideal solution for you.

    If you do choose to use an alternative malloc library make sure that
you have a license to do so and that you follow any distribution

requirements. On systems that support dynamic linking you may want to
link the library statically rather than dynamically so that you don't
have to worry about an additional file that would need to be installed.
However, whether you have that choice depends on the license for the
specific library, and some licenses also require that the source code
for the library be made readily available. Shared libraries have the
advantage that they can be updated with bug fixes so that all programs
that require these libraries will automatically receive these fixes
without needing to be relinked.

    If all of the above suggestions do not seem to help and you still
feel that you have a performance bottleneck in the part of your code
that deals with dynamically allocated memory then you should try using
the memory allocation profiling feature of mpatrol. This can be used
at run-time to analyse the dynamic memory allocation calls that your
program makes during its execution, and builds statistics for later
viewing with the mprof command. It is then possible for you to see
exactly how many calls were made to each function and where they came
from. Such information can then be put to good use in order to
optimise the relevant parts of your code.

    And finally, some tips on how to correctly use dynamic memory
allocations. The first, most basic rule is to always check the return
values from malloc() and related functions. Never assume that a call to
malloc() will succeed, because you're unlikely to be able to read the
future(3). Alternatively, use (or write) an xmalloc() or similar
function, which calls malloc() but never returns NULL since it will
abort instead. With the C++ operators it is slightly different because
some versions use exceptions to indicate failure, so you should always
provide a handler to deal with this eventuality.

    Never use features(4) of specific malloc libraries if you want your
code to be portable. Always follow the ANSI C or C++ calling
conventions and never make assumptions about the function or operator
you are about to call -- the standards committees went to great lengths
to explicitly specify its behaviour. For example, don't assume that
the contents of a freed memory allocation will remain valid until the
next call to malloc(), and don't assume that the contents of a newly
allocated memory block will be zeroed unless you created it with
calloc().

    Finally, try stress-testing your program in low memory conditions.
The mpatrol library contains the LIMIT option which can place an upper
bound on the size of the heap, and also contains the FAILFREQ and
FAILSEED options which can cause random memory allocation failures.
Doing this will test parts of your code that you would probably never
expect to be called, but perhaps they will one day! Who would you
rather have debugging your program -- yourself or the user?

    ---------- Footnotes ----------

    (1) Such as for use in a linked list.

    (2) A freely distributably library called GC (see
            Related software
).

(3) If you can, why are you reading this -- you've already read it!

(4) Whether they are documented or not.

1.26 mpatrol.guide/How it works

```
How it works
************
```

   The mpatrol library was originally written with the intention of
plugging it into an existing compiler so that the compiler could plant
calls to it in the code it generated when a specific debugging option
was used. These extra calls would obviously slow the code down, but
along with the stack checking options that would be provided, this
would give the user an enhanced run-time debugging environment.
Unfortunately, this integration never happened, but the way that
mpatrol works is still significantly different from other malloc tracing
libraries.

   In order to quickly determine exactly which memory allocation a heap
address belonged to it was necessary to be able to search the heap in
an efficient manner. The traditional way of searching along a linked
list was unfeasible, so an implementation based on red-black trees was
used, where every known memory allocation in the heap was given an
entry in the tree, with their start addresses as the key. Another
major design decision was to also choose red-black trees to implement
the best fit allocation algorithm. Although first fit was considered,
I decided that best fit would allow the library to have more control
over the heap, with every free memory block in the heap given an entry
in the free tree, with their sizes as the key. There was a bit of work
involved in getting the splitting and merging of free blocks to work
efficiently, but it seems to work well now.

   My original implementation had all of the information about each
memory block stored just before the block itself. I eventually dropped
that behaviour in favour of storing all of the library's internal
information in a separate part of the heap. I did that for two
reasons. The first was because of the problems that would occur due to
memory allocations with different alignment requirements. The second
reason was that the library's internal structures could be
write-protected on systems with virtual memory, to prevent user code
interfering with the operation of the library.

   Because the library attempts to record as much information as
possible about every memory allocation there will inevitably be a much
larger memory requirement when running a program linked with the
library. This will typically be two or three times larger in
magnitude, but will be affected by the number of memory allocations
made and also the number of symbols read. The latter will also affect
how quickly the program starts since the first call to allocate memory
will result in the initialisation of the library and the loading of
symbols from the executable file and any shared libraries.

Due to its design, it is also possible to allocate memory from the
heap using the mpatrol library functions whilst already within an
mpatrol library function. This does not normally occur, but on some
platforms calling printf() from within the library may result in
printf() calling malloc() to allocate itself a buffer, which ends up as
a recursive call. Luckily, this is dealt with by simply not displaying
the allocation in the log file, but all other details of the allocation
are still recorded. This can sometimes result in hidden memory usage
which occurs behind the scenes and alters the peak memory usage in the
summary. This is particularly evident when the library uses an object
file access library to read program symbols at the time of library
initialisation.

Memory allocation profiling support was added for mpatrol release
1.2.0. Every allocation and deallocation is recorded, with the call
stack information being used to differentiate all of the call sites
within the program. Unlike other profilers that come with UNIX
systems, even the symbolic information about the program being run is
written to the profiling output file, since it makes no sense for mprof
to re-read the symbol table from the executable file when it has
already been read and processed by the mpatrol library. It also has
the added bonus of allowing the user to save profiling output files for
later use even when the executable files which produced them have
changed or no longer exist.

The library is written in a modular fashion so as to make it easy to
add new functionality. New modules have already been added, such as
the stack, symbol and profile modules. Extra information about each
memory allocation can be added to the allocation information module in
src/info.h and src/info.c without having to change much code in any
other files.

1.27 mpatrol.guide/Examples

                 Examples
****

Following are a set of examples that are intended to illustrate what
exactly is possible with the mpatrol library and how to go about using
it effectively.

You should already have built and installed the library and should
know how to link programs with the library. Unfortunately, it isn't
possible to give specific instructions on how to do this as it varies
from system to system and also depends on your preferred compiler and
development tools.

However, on a typical SVR4 UNIX system, with mpatrol installed in
/usr/local, the mpatrol library can usually be incorporated into a
program using the following commands:

  * If the mpatrol library was built with no support for any object
    file format or was built with support for the COFF object file
    format:

```
        cc -I/usr/local/include <file> -L/usr/local/lib -lmpatrol
```

  * If the mpatrol library was built with support for the ELF32 object
    file format access library:

```
        cc -I/usr/local/include <file> -L/usr/local/lib -lmpatrol -lelf
```

  * If the mpatrol library was built with support for the GNU BFD
    object file format access library:

```
        cc -I/usr/local/include <file> -L/usr/local/lib -lmpatrol -lbfd
            -liberty
```

   If you need to link with other libraries, make sure that they don't
contain definitions of malloc(), etc., or if they do then you must
ensure that the mpatrol library appears before them on the link line.

   You should also know how to set an environment variable on your
specific system. Again, this varies from system to system and also
depends on the command line interpreter or shell that you use. The
environment variable that the mpatrol library uses is called
MPATROL_OPTIONS. You can see exactly what options are available for
this environment variable by setting it to HELP and then running a
program that has been linked with the library.

                Example 1
                 Getting started.

                Example 2
                 Detecting incorrect reuse of freed memory.

                Example 3
                 Detecting use of free memory.

                Example 4
                 Using overflow buffers.

                Example 5
                 Bad memory operations.

                Example 6
                 Incompatible function calls.

                Example 7
                 Additional useful information.

1.28   mpatrol.guide/Example 1

```
Getting started
===============
```

The first example we'll look at is when the argument in a call to
free() doesn't match the return value from malloc(), even though the
intention is to free the memory that was allocated by malloc(). This
example is in tests/fail/test1.c and causes many existing malloc()
implementations to crash.

Along the way, I'll try to describe as many features of the mpatrol
library as possible, and illustrate them with examples. Note that the
output from your version of the library is likely to vary slightly from
that shown in the examples, especially on non-UNIX systems.

```
23  /*
24   * Allocates a block of 16 bytes and then attempts to free the
25   * memory returned at an offset of 1 byte into the block.
26   */


29  #include "mpatrol.h"


32  int main(void)
33  {
34      char *p;

36      if (p = (char *) malloc(16))
37          free(p + 1);
38      return EXIT_SUCCESS;
39  }
```

Note that I've removed the copyright message from the start of the
file and added line numbers so that the tracing below makes more sense.

After compiling and linking the above program with the mpatrol
library, the MPATROL_OPTIONS environment variable should be set to be
LOGALL and the program should be executed, generating the following
output in mpatrol.log.

```
@(#) mpatrol 1.2.0 (00/05/16)
Copyright (C) 1997-2000 Graeme S. Roy

This is free software, and you are welcome to redistribute it under
certain conditions; see the GNU Library General Public License for
details.

For the latest mpatrol release and documentation,
visit http://www.cbmamiga.demon.co.uk/mpatrol.

Log file generated on Tue May  2 23:41:04 2000

ALLOC: malloc (13, 16 bytes, 8 bytes) [main|test1.c|36]
        0x00010AE0 main
        0x000109D4 _start

returns 0x00028000

FREE: free (0x00028001) [main|test1.c|37]
```

```
              0x00010B24 main
              0x000109D4 _start

    ERROR: free: 0x00028001 does not match allocation of 0x00028000
        0x00028000 (16 bytes) {malloc:13:0} [main|test1.c|36]
              0x00010AE0 main
              0x000109D4 _start

    system page size:  8192 bytes
    default alignment: 8 bytes
    overflow size:     0 bytes
    overflow byte:     0xAA
    allocation byte:   0xFF
    free byte:         0x55
    allocation stop:   0
    reallocation stop: 0
    free stop:         0
    unfreed abort:     0
    small boundary:    32
    medium boundary:   256
    large boundary:    2048
    lower check range: -
    upper check range: -
    failure frequency: 0
    failure seed:      533453
    prologue function: <unset>
    epilogue function: <unset>
    handler function:  <unset>
    log file:          mpatrol.log
    profiling file:    mpatrol.out
    program filename:  ./test1
    symbols read:      3240
    autosave count:    0
    allocation count:  13
    allocation peak:   4720 bytes
    allocation limit:  0 bytes
    allocated blocks:  1 (16 bytes)
    freed blocks:      0 (0 bytes)
    free blocks:       1 (8176 bytes)
    internal blocks:   25 (204800 bytes)
    total heap usage:  212992 bytes
    total compared:    0 bytes
    total located:     0 bytes
    total copied:      0 bytes
    total set:         0 bytes
    total warnings:    0
    total errors:      1
```

   Ignoring the copyright blurb at the top, let's first take a look at
the initial log message from the library. I've annotated each of the
items with a number that corresponds to the descriptions below.

```
    (1)     (2)    (3)      (4)             (5)         (6)      (7)    (8)
     |       |      |        |               |           |        |      |
     V       V      V        V               V           V        V      V
    ALLOC: malloc (13, 16 bytes, 8 bytes) [main|test1.c|36]
      (9) -> 0x00010AE0 main
```

```
        0x000109D4 _start <- (10)

    returns 0x00028000 <- (11)
```

1. Allocation type. This generalises the type of dynamic memory
   operation that is being performed, and can be one of ALLOC,
   REALLOC or FREE. This should make looking for all allocations,
   reallocations or frees in the log file a lot easier.
   Alternatively, if a memory operation function was called then this
   can also be one of MEMSET, MEMCOPY, MEMFIND or MEMCMP.

2. Allocation function. This is the name of the function that has
   been called to allocate the memory, in this case malloc.

3. Allocation index. This is incremented every time a new memory
   allocation is requested, and persists even if the memory
   allocation is resized with realloc(), recalloc() or expand(), so
   can be useful to keep track of a memory allocation, even if its
   start address changes. The mpatrol library may use up the first
   few allocation indices when it gets initialised.

4. Size of requested allocation.

5. Alignment for requested allocation. This is normally the default
   system alignment for general-purpose memory allocations, but may
   be different depending on the type of function that is used to
   allocate the memory.

   The following information contains source file details of where the
call to malloc() came from, but is only available if the source file
containing the call to malloc() included mpatrol.h; otherwise the
fields will all be -(1). Because of the convoluted way this
information is obtained for the C++ operators, you may encounter some
problems in existing C++ programs when making direct calls to operator
new for example. However, if you want to disable the redefinition of
the C++ operators in mpatrol.h you can define the preprocessor macro
MP_NOCPLUSPLUS before the inclusion of that file.

6. Function where call to malloc() took place. This information is
   only available if the source file containing the call to malloc()
   was compiled with gcc or g++.

7. Filename in which call to malloc() took place.

8. Line number at which call to malloc() took place.

   The following information contains function call stack details of
where the call to malloc() came from, but is only available if the
mpatrol library has been built on a platform that supports this. The
top-most entry should be the function which called malloc() and the
bottom-most entry should be the entry-point for the process.

9. Address of function call. This is normally the address of the
   machine instruction immediately after the function call
   instruction, also known as the return address.

10. Function where call took place. This information is only

available if the mpatrol library has been built on a platform that
supports reading symbol table information from executable files,
and then only if there is an entry in the symbol table
corresponding to the return address. C++ function names may still
be in their mangled form, but this can be easily rectified by
processing the log file with a C++ name demangler.

The following information is only available when the allocation type
is ALLOC or REALLOC since it makes no sense when applied to FREE.

 11. The address of the new memory block that has been allocated by
     malloc().

As you can see, there is quite a lot of information that can be
displayed from a simple call to malloc(), and hopefully this
information has been presented in a clear and concise format in the log
file.

The next entries in the log file correspond to the call to free(),
which attempts to free the memory allocated by malloc(), but supplies
the wrong address.

The first three lines should be self-explanatory as they are very
similar to those described above for malloc(). However, the next lines
signal that a terminal error has occurred in the program, so I've
annotated them as before.

```
    FREE: free (0x00028001) [main|test1.c|37]
          0x00010B24 main
          0x000109D4 _start


     (1)    (2)
      |      |
      V      V
    ERROR: free: 0x00028001 does not match allocation of 0x00028000
          (3)           (4)         (5)  (6)(7)  (8)      (9)   (10)
           |             |           |    | |    |        |     |
           V             V           V    V V    V        V     V
       0x00028000 (16 bytes) {malloc:13:0} [main|test1.c|36]
    (11) -> 0x00010AE0 main
          0x000109D4 _start
```

  1. Error severity. The mpatrol library has two different severities
     of error: WARNING and ERROR. The first is always recoverable, and
     serves only to indicate that something is not quite right, and so
     may be useful in determining where something started to go wrong.
     The second may or may not be recoverable, and the library
     terminates the program if it is fatal, displaying any relevant
     information as it does this.

  2. Allocation function. This is the name of the function used to
     allocate, reallocate or free memory where the error was detected.
     This may be omitted if an error is detected elsewhere in the
     library.

The following information is related to the information that the
library has stored about the relevant memory allocation. This

information is always displayed in this format when details of
individual memory allocations are required. If any information is
missing then it simply means that the library was not able to determine
it when the memory block was first allocated.

   3. Address of memory allocation.

   4. Size of memory allocation.

   5. Allocation function. This is the name of the function that was
      called to allocate the memory block, in this case malloc. If the
      memory allocation has been resized then this will be either
      realloc, recalloc or expand.

   6. Allocation index.

   7. Reallocation index. This is used to count the number of times a
      memory allocation has been resized with realloc(), recalloc() or
      expand().

   8. Function where original call to malloc() took place. If the memory
      allocation has been resized then this will be the name of the
      function which last called realloc(), recalloc() or expand().

   9. Filename in which original call to malloc() took place. If the
      memory allocation has been resized then this will be the filename
      in which the last call to realloc(), recalloc() or expand() took
      place.

  10. Line number at which original call to malloc() took place. If the
      memory allocation has been resized then this will be the line
      number at which the last call to realloc(), recalloc() or expand()
      took place.

  11. Function call stack of original memory allocation. If the memory
      allocation has been resized then this will be the call stack of
      the last call to realloc(), recalloc() or expand().

   So, the mpatrol library detected the error in the above program and
terminated it. When the library terminates it always displays a
summary of various memory allocation statistics and settings that were
used during the execution of the program.

   The various settings and statistics displayed by the library for the
above example have been numbered and their descriptions appear below.

```
     1  system page size:  8192 bytes
     2  default alignment: 8 bytes
     3  overflow size:     0 bytes
     4  overflow byte:     0xAA
     5  allocation byte:   0xFF
     6  free byte:         0x55
     7  allocation stop:   0
     8  reallocation stop: 0
     9  free stop:         0
    10  unfreed abort:     0
    11  small boundary:    32
```

```
12  medium boundary:    256
13  large boundary:     2048
14  lower check range:  -
15  upper check range:  -
16  failure frequency:  0
17  failure seed:       533453
18  prologue function:  <unset>
19  epilogue function:  <unset>
20  handler function:   <unset>
21  log file:           mpatrol.log
22  profiling file:     mpatrol.out
23  program filename:   ./test1
24  symbols read:       3240
25  autosave count:     0
26  allocation count:   13
27  allocation peak:    4720 bytes
28  allocation limit:   0 bytes
29  allocated blocks:   1 (16 bytes)
30  freed blocks:       0 (0 bytes)
31  free blocks:        1 (8176 bytes)
32  internal blocks:    25 (204800 bytes)
33  total heap usage:   212992 bytes
34  total compared:     0 bytes
35  total located:      0 bytes
36  total copied:       0 bytes
37  total set:          0 bytes
38  total warnings:     0
39  total errors:       1
```

1. System page size. This value is used on some platforms when
   allocating and protecting system memory.

2. Default alignment. This value is the minimum alignment required
   for general purpose memory allocations, and is usually the
   alignment required by the most restrictive datatype on a given
   system. It is used when allocating memory that has no specified
   alignment. It can be changed at run-time using the DEFALIGN
   option, but setting this value too small may cause the program to
   crash due to bus errors which are caused by reading from or
   writing to misaligned data.

3. Overflow size. This value is the size used by one overflow
   buffer. If this is non-zero then every memory allocation will
   have two overflow buffers; one on either side. These buffers are
   used by the library to detect if the program has written too many
   bytes to a memory allocation, thus overflowing into one of the
   buffers, but these extra checks can slow down execution speed. It
   can be changed at run-time using the OFLOWSIZE option.

4. Overflow byte.

5. Allocation byte.

6. Free byte. These values are used by the library to pre-fill
   blocks of memory for checking purposes. The overflow byte is used
   to fill overflow buffers, the allocation byte is used to fill
   newly-allocated memory (except from calloc() or recalloc()), and

the free byte is used to fill free blocks or freed memory
allocations. These can be changed at run-time using the
OFLOWBYTE, ALLOCBYTE and FREEBYTE options.

7. Allocation stop.

8. Reallocation stop.

9. Free stop. These values are used by the library to halt the
   program when run inside a debugger whenever a specified allocation
   index is allocated, reallocated or freed. These can be changed at
   run-time using the ALLOCSTOP, REALLOCSTOP and FREESTOP options.

10. Unfreed abort. This value is used when the program terminates and
    is used by the library to check if there are more than a given
    number of unfreed memory allocations. If there are then the
    library will cause the program to abort with an error. It can be
    changed at run-time using the UNFREEDABORT option.

11. Small boundary.

12. Medium boundary.

13. Large boundary. These values are used in memory allocation
    profiling and specify the boundaries in bytes between small,
    medium, large and extra large allocations. These can be changed
    at run-time using the SMALLBOUND, MEDIUMBOUND and LARGEBOUND
    options.

14. Lower check range.

15. Upper check range. These values specify the range of allocation
    indices through which the library will physically check every area
    of free memory and every overflow buffer for errors. A dash
    specifies that either the lower or upper range is infinite, but if
    they are both zero then no such checking will ever be performed,
    thus speeding up execution speed dramatically. The library
    defaults to performing checks for every allocation index. These
    can be changed at run-time using the CHECK option.

16. Failure frequency.

17. Failure seed. These values are used to specify if random memory
    allocation failures should occur during program execution, for the
    purposes of stress testing a program. If the failure frequency is
    zero then no random failures will occur, but if it is greater than
    zero then the higher the number, the less frequent the failures.
    The failure seed is used internally by the mpatrol library when
    generating random numbers. If it is zero then the seed will be set
    randomly, but if it is greater than zero then it will be used to
    generate a predictable sequence of random numbers; i.e. two runs
    of the same program with the same failure frequencies and the same
    failure seeds will generate exactly the same sequence of failures.

18. Prologue function.

19. Epilogue function.

20. Handler function. These values contain addresses or names of
    functions that have been installed as callback functions for the
    library. These functions, if set, will be called from the library
    at appropriate times during program execution in order to handle
    specific events. These can be changed at compile-time using the
    __mp_prologue(), __mp_epilogue() and __mp_nomemory() functions.

21. Log file. Simply contains the name of the file where all mpatrol
    library diagnostics go to. It can be changed at run-time using
    the LOGFILE option.

22. Profiling file. Contains the name of the file where all of the
    mpatrol library memory allocation profiling information goes when
    the PROF option is used. It can be changed at run-time using the
    PROFFILE option.

23. Program filename. Contains the full pathname to the program's
    executable file. This is used by the mpatrol library to read the
    symbol table in order to provide symbolic information in function
    call stacks. It can be changed at run-time using the PROGFILE
    option.

24. Symbols read. This value contains the total number of symbols
    read from a program's executable file and/or the dynamic linker,
    if applicable.

25. Autosave count. This value contains the frequency at which the
    mpatrol library should 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. It can be changed at run-time using the
    AUTOSAVE option.

26. Allocation count. This value contains the total number of memory
    allocations that were created by the mpatrol library. This value
    may be more than expected if the mpatrol library makes any memory
    allocations during initialisation.

27. Allocation peak. This value contains the peak memory usage set by
    the program when running. This value may be more than expected if
    the mpatrol library makes any memory allocations during
    initialisation.

28. Allocation limit. This value is used to limit the amount of
    memory that can be allocated by a program, which can be useful for
    stress-testing in simulated low memory conditions. It can be
    changed at run-time using the LIMIT option.

29. Allocated blocks.

30. Freed blocks.

31. Free blocks. These values contain the total number of allocated,
    freed and free blocks at the time the summary was produced. A
    freed block is an allocated block that has been freed but has not

been returned to the free memory list for later allocation. These
values may be different from those expected if the mpatrol library
makes any memory allocations during initialisation.

32. Internal blocks. This value contains the total number of memory
    blocks (of varying sizes) that have been allocated from the system
    for the mpatrol library to use internally. These memory blocks
    will be write-protected on systems that support memory protection
    in order to prevent the program from corrupting the library's data
    structures. This can be overridden at run-time using the
    NOPROTECT option in order to speed up program execution slightly.

33. Total heap usage. This value contains the total amount of system
    heap memory that has been allocated by the mpatrol library.

34. Total compared.

35. Total located.

36. Total copied.

37. Total set. These values contain the total number of bytes that
    have been tracked by the mpatrol library in byte comparison
    operations (such as memcmp()), byte location operations (such as
    memchr()), byte copy operations (such as memcpy()) and byte set
    operations (such as memset()) respectively. They do not take into
    account any other such operations that occur outwith these
    functions, such as loading and storing from machine instructions.

38. Total warnings.

39. Total errors. The library keeps a count of the total number of
    warnings and errors it has displayed so that you can quickly work
    out this information at program termination.

    ---------- Footnotes ----------

    (1) This information may also be filled in if the USEDEBUG option is
used and supported, and if debugging information about the call to
malloc() is available.

1.29 mpatrol.guide/Example 2

Detecting incorrect reuse of freed memory
===

   The next example uses tests/fail/test2.c to illustrate how the
mpatrol library can detect whereabouts on the heap an address belongs.

     23 /*
     24   * Allocates a block of 16 bytes and then immediately frees it. An
     25   * attempt is then made to double the size of the original block.
     26   */

```
29  #include "mpatrol.h"


32  int main(void)
33  {
34      char *p;

36      if (p = (char *) malloc(16))
37      {
38          free(p);
39          p = (char *) realloc(p, 32);
40      }
41      return EXIT_SUCCESS;
42  }
```

The relevant excerpts from mpatrol.log appear below. The format of
the log messages should be familiar to you now.

```
ALLOC: malloc (13, 16 bytes, 8 bytes) [main|test2.c|36]
        0x00010B18 main
        0x00010A0C _start

returns 0x00028000

FREE: free (0x00028000) [main|test2.c|38]
        0x00010B54 main
        0x00010A0C _start

    0x00028000 (16 bytes) {malloc:13:0} [main|test2.c|36]
        0x00010B18 main
        0x00010A0C _start

REALLOC: realloc (0x00028000, 32 bytes, 8 bytes) [main|test2.c|39]
        0x00010B88 main
        0x00010A0C _start

ERROR: realloc: 0x00028000 has not been allocated

returns 0x00000000
```

The mpatrol library stores all of its information about allocated
and free memory in tree structures so that it can quickly determine if
an address belongs to allocated or free memory, or if it even exists in
the heap that is managed by mpatrol. The above example should
illustrate this since after the allocation had been freed, the library
recognised this and reported an error. It was possible for the program
to continue execution even after that error since mpatrol could recover
from it and return NULL.

It is possible for mpatrol to give even more useful diagnostics in
the above situation by using the NOFREE option. This prevents the
library from returning any freed allocations to the free memory pool,
by preserving any information about them and marking them as freed. If
you add the NOFREE option to the MPATROL_OPTIONS environment variable
you should see the following entries in mpatrol.log instead.

```
      ALLOC: malloc (13, 16 bytes, 8 bytes) [main|test2.c|36]
              0x00010B18 main
              0x00010A0C _start

      returns 0x00029DE0

      FREE: free (0x00029DE0) [main|test2.c|38]
              0x00010B54 main
              0x00010A0C _start

          0x00029DE0 (16 bytes) {malloc:13:0} [main|test2.c|36]
              0x00010B18 main
              0x00010A0C _start

      REALLOC: realloc (0x00029DE0, 32 bytes, 8 bytes) [main|test2.c|39]
              0x00010B88 main
              0x00010A0C _start

      ERROR: realloc: 0x00029DE0 was freed with free
          0x00029DE0 (16 bytes) {free:13:0} [main|test2.c|38]
              0x00010B54 main
              0x00010A0C _start

      returns 0x00000000
```

   Note the extra information reported by realloc() since the library
knows all of the details about the freed memory allocation and when it
was freed.

   The NOFREE option tends to use up much more system memory than normal
since it effectively instructs the mpatrol library to allocate new
memory for every single memory allocation or reallocation. It can also
slow down program execution when overflow buffers are used, since with
each new memory allocation the library needs to check more and more
overflow buffers every time it is called. However, it can be quite
useful for problems such as this one. The test in tests/fail/test3.c
has a similar situation.

   Normally, the NOFREE option will cause the library to fill all freed
memory allocations with the free byte. However, the original contents
of such allocations can be preserved with the PRESERVE option. This
could help in situations when you need to determine exactly if a
program is relying on the contents of freed memory.

1.30   mpatrol.guide/Example 3

```
Detecting use of free memory
============================
```

   This next example illustrates how the mpatrol library is able to
check to see if anything has been written into free memory. The test
is located in tests/fail/test4.c and simply writes a single byte into
free memory.

```
23  /*
24   * Allocates a block of 16 bytes and then immediately frees it.  A
25   * NULL character is written into the middle of the freed memory.
26   */


29  #include "mpatrol.h"


32  int main(void)
33  {
34      char *p;

36      if (p = (char *) malloc(16))
37      {
38          free(p);
39          p[8] = '\0';
40      }
41      return EXIT_SUCCESS;
42  }
```

   The following output was produced as part of mpatrol.log. Note that
this test was run using the same MPATROL_OPTIONS settings as the last
example, but make sure that PRESERVE is not set.

```
    ERROR: freed allocation 0x00029DE0 has memory corruption at 0x00029DE8
           0x00029DE8  00555555 55555555                       .UUUUUUU

       0x00029DE0 (16 bytes) {free:13:0} [main|test4.c|38]
           0x00010B1C main
           0x000109D4 _start
```

   The library was able to detect that something had been written into
free memory and could report on the memory allocation that was
overwritten. However, these checks are only performed whenever a
function in the mpatrol library is called. In the example above, the
code which wrote into free memory could have been miles away from where
the library detected the error.

   On platforms that support memory protection, the library also
supports the PAGEALLOC option. This option instructs the library to
force every single memory allocation to have a size which is a multiple
of the system page size. Although the library still stores the
original requested size, it effectively means that no two memory
allocations occupy the same page of memory. It can then use page
protection (which only operates on pages of memory) to protect all free
memory from being read from or written to, and uses similar features to
install a page of overflow buffer on either side of the allocation.

   However, if the requested size for the memory allocation was not a
multiple of the page size this means that there will still be unused
space left over in the allocated pages. This problem is solved by
turning the unused space into overflow buffers that will be checked in
the normal way. The positioning of the allocation within its pages is
also important. If you want to check for illegal reads from the
borders of the memory allocation, unless it fits exactly into its pages
then there is a chance that a program could illegally read the

right-most overflow buffer if the allocation was left-aligned, or
vice-versa. Two settings therefore exist for the PAGEALLOC option:
LOWER and UPPER. They refer to the placement of every memory
allocation within its constituent pages.

   The following diagram illustrates the PAGEALLOC option. In the
diagram, the system page size is assumed to be 16 bytes (very unlikely,
but will serve for this example) and each character represents 1 byte.

```
    x = allocated memory
    o = overflow buffer (filled with the overflow byte)
    . = overflow buffer page (read and write protected)

    PAGEALLOC=LOWER, allocation size is 16 bytes or
    PAGEALLOC=UPPER, allocation size is 16 bytes:
        ................xxxxxxxxxxxxxxxx................

    PAGEALLOC=LOWER, allocation size is 8 bytes:
        ................xxxxxxxxoooooooo................

    PAGEALLOC=UPPER, allocation size is 8 bytes:
        ................ooooooooxxxxxxxx................
```

   In our original example, if the PAGEALLOC=LOWER option is added to
the MPATROL_OPTIONS environment variable then the following error will
be produced instead of the original error.

```
    ERROR: illegal memory access at address 0x0009E008
        0x0009E000 (16 bytes) {free:13:0} [main|test4.c|38]
            0x00010B1C main
            0x000109D4 _start

        call stack
            0x00010B1C main
            0x000109D4 _start
```

   On systems that support memory protection, the mpatrol library has a
built-in signal handler which catches illegal memory accesses and
terminates the program. In the above case, the freed memory was made
write-protected and so could not be written to. The underlying virtual
memory system in the operating system noticed this and signaled this to
the library immediately after it happened.

   Along with the details of the freed memory allocation that was being
written to, the library also attempts to display the function call
stack for the location in the program that caused the illegal memory
access, although this can be quite unreliable. A better solution would
be to run the program in a debugger to catch the illegal memory access.

   Note that the PAGEALLOC option also modifies the behaviour of the
NOFREE and PRESERVE options when used together. The memory allocation
being freed will always be made write-protected when the PRESERVE
option is used, otherwise it will also be made read-protected to
prevent further accesses.

   Note also that the PAGEALLOC=UPPER option is potentially much less
efficient at catching illegal memory accesses than the PAGEALLOC=LOWER

option. This is due to alignment requirements, since an allocation of
1 byte requiring an alignment of 16 bytes cannot be placed at the very
end of a page of size 4096 bytes. The following diagram illustrates
this, using the same page size as the last diagram.

```
    x = allocated memory
    o = overflow buffer (filled with the overflow byte)
    . = overflow buffer page (read and write protected)

    PAGEALLOC=UPPER, allocation size is 16 bytes, alignment is 8 bytes:
        ................xxxxxxxxxxxxxxxx................

    PAGEALLOC=UPPER, allocation size is 3 bytes, alignment is 1 byte:
        ................ooooooooooooooxxx................

    PAGEALLOC=UPPER, allocation size is 3 bytes, alignment is 8 bytes:
        ................ooooooooxxxooooo................
```

Everything is OK until the last allocation, where the alignment
requirement means that there must be two overflow buffers. This slows
down program execution since the library must check an additional
overflow buffer, and also means that the program would have to read six
bytes beyond the end of the allocation before the illegal memory access
would be detected.

1.31 mpatrol.guide/Example 4

```
Using overflow buffers
======================
```

This example illustrates the use of overflow buffers and so the
MPATROL_OPTIONS environment variable should have OFLOWSIZE=2 added to
it. However, turn off any PAGEALLOC options for the purposes of this
example. The test is located in tests/fail/test5.c, and
tests/fail/test6.c is very similar.

```
    23  /*
    24   * Allocates a block of 16 bytes and then copies a string of 16
    25   * bytes into the block.  However, the string is copied to 1 byte
    26   * before the allocated block which writes before the start of the
    27   * block.  This test must be run with an OFLOWSIZE greater than 0.
    28   */


    31  #include "mpatrol.h"


    34  int main(void)
    35  {
    36      char *p;

    38      if (p = (char *) malloc(16))
    39      {
    40          strcpy(p - 1, "this test fails!");
```

```
    41              free(p);
    42          }
    43          return EXIT_SUCCESS;
    44      }
```

   The following error should be produced in mpatrol.log.

   ERROR: allocation 0x00029E28 has a corrupted overflow buffer at 0x00029E27
          0x00029E26 AA74   ªt

       0x00029E28 (16 bytes) {malloc:13:0} [main|test5.c|38]
          0x00010B0C main
          0x00010A00 _start

   Once again, the library attempts to show you as much detail as
possible about where the corruption occurred. Along with showing you a
memory dump of the overflow buffer that was corrupted, it also shows
you the allocation to which the overflow buffer belongs.

   Using overflow buffers can reduce the speed of program execution
since the library has to check every buffer whenever it is called, and
if the buffers are larger then they'll take longer to check and will
use up more memory. However, larger buffers mean that there is less
chance of the program writing past one memory allocation into another.

   Alternatively, the CHECK option can be used to limit the number of
checks that the library has to perform, thus speeding up program
execution. This option specifies a range of allocation indices through
which the library will check overflow buffers and free memory for
corruption. Such checks occur when they normally would, but only if
the current allocation index falls within the specified range. This
feature can be used when there is a suspicion that free memory
corruption or overflow buffer corruption occurs at a certain point
during program execution, but checking them at every library call would
take too long.

   On systems which support software watch points, there is an extra
option called OFLOWWATCH which allows additional memory protection.
Watch points allow individual bytes to be read and/or write protected
as opposed to just pages. The OFLOWWATCH option installs software
watch points at every overflow buffer instead of requiring the library
to check the integrity of the overflow buffers, and can be used in
combination with PAGEALLOC. However, software watch points slow down
program execution to a crawl since every machine instruction must be
checked individually by the system to see if it accesses a watch point
area. Slowing the program down by a factor of 10,000 is not uncommon
on some systems when the OFLOWWATCH option is used.

1.32 mpatrol.guide/Example 5

```
Bad memory operations
=====================
```

   In C there are several basic memory operation functions that are

often called to perform tasks such as clearing memory, copying memory,
etc. The mpatrol library contains replacements for these which allow
for better checking of their arguments to prevent reading and writing
past the boundaries of existing memory allocations. The following
source can be found in tests/fail/test9.c.

```
23  /*
24   * Allocates a block of 16 bytes and then attempts to zero the contents  ←
     of
25   * the block.  However, a zero byte is also written 1 byte before and 1
26   * byte after the allocated block, resulting in an error in the log file.
27   */


30  #include "mpatrol.h"


33  int main(void)
34  {
35      char *p;

37      if (p = (char *) malloc(16))
38      {
39          memset(p - 1, 0, 18);
40          free(p);
41      }
42      return EXIT_SUCCESS;
43  }
```

   When this is compiled and run, the following should appear in the
log file.

```
ERROR: memset: range [0x00027FFF,0x00028010] overflows [0x00028000,0x0002800F ←
    ]
   0x00028000 (16 bytes) {malloc:13:0} [main|test9.c|37]
        0x00010B18 main
        0x00010A0C _start
```

   As you can see, the library detected that the memset() function would
have written past the boundaries of the memory allocation and reported
this to you. It then proceeded to ignore the request to copy the
memory and continued with the execution of the program(1). Note that
this will only be done for known memory allocations. Reading and
writing past the boundaries of static and stack memory allocations
cannot be detected in this way.

   If the LOGMEMORY option is added to the MPATROL_OPTIONS environment
variable then it is possible to see a log of all the mpatrol library
memory operation functions that were called during program execution.
For example, adding this option and running the above program again
will produce something similar to the following.

```
MEMSET: memset (0x00027FFF, 18 bytes, 0x00) [main|test9.c|39]
        0x00010B18 main
        0x00010A0C _start
```

   This is similar to the tracing produced for memory allocation

functions, except that the arguments in parentheses mean different
things. For MEMSET, the first argument represents the start of the
memory block to set, the second argument represents the number of bytes
to set and the third argument represents the actual byte to set.

   For MEMCOPY, the first argument represents the source memory block,
the second argument represents the destination memory block, the third
argument represents the number of bytes to copy and the fourth argument
represents a byte to copy up to if memccpy() is being called. This is
similar for MEMCMP.

   For MEMFIND, the first and second arguments represent the source
memory block and its length, while the third and fourth arguments
represent the memory block to search for and its length. In the
implementation for memchr(), the byte to search for is copied to a one
byte buffer and the address of that buffer is used as the memory block
to search for.

   Note that as with the memory allocation functions, MEMCMP, MEMFIND,
MEMCOPY and MEMSET are used to generalise the types of operations being
performed and are followed by the names of the actual functions being
used. In some cases the functions may use a different ordering of
parameters than that shown.

        ---------- Footnotes ----------

   (1) The error can be turned into a warning with the ALLOWOFLOW
option which will also force the operation to be performed.

1.33   mpatrol.guide/Example 6

Incompatible function calls
===========================

   This example illustrates how the mpatrol library checks for calls to
incompatible pairs of memory allocation functions. It requires the use
of C++, although does not use any C++ features except for overloaded
operators. The source is in tests/fail/test7.c, and tests/fail/test8.c
is similar.

```
   23  /*
   24   * Allocates a block of 16 bytes using C++ operator new[] and then
   25   * attempts to free it using C++ operator delete.
   26   */


   29  #include "mpatrol.h"


   32  int main(void)
   33  {
   34      char *p;

   36      p = new char[16];
```

```
37      delete p;
38      return EXIT_SUCCESS;
39  }
```

The relevant parts of mpatrol.log are shown below.

```
ALLOC: operator new[] (17, 16 bytes, 8 bytes) [int main()|test7.c|36]
        0x00010A28 __builtin_vec_new
        0x00010ADC main
        0x000108D0 _start

returns 0x00028000

FREE: operator delete (0x00028000) [int main()|test7.c|37]
        0x00010A74 __builtin_delete
        0x00010AF0 main
        0x000108D0 _start

ERROR: operator delete: 0x00028000 was allocated with operator new[]
    0x00028000 (16 bytes) {operator new[]:17:0} [int main()|test7.c|36]
        0x00010A28 __builtin_vec_new
        0x00010ADC main
        0x000108D0 _start
```

This shows a call to operator new[], closely followed by a call to operator delete. However, in C++ calls to operator new[] must be matched by calls to operator delete[] and not operator delete. Hence, the library reports this as an error and does not free the memory allocation.

1.34   mpatrol.guide/Example 7

```
Additional useful information
============================
```

This last example illustrates the various SHOW options that are available for displaying additional information from the mpatrol library at program termination. It also shows how to easily detect memory leaks. Use the OFLOWSIZE=16, NOFREE and SHOWALL options in MPATROL_OPTIONS before running.

```
 1  /*
 2   * Introduces a memory leak by clobbering a pointer with a new
 3   * memory allocation.  Use with SHOWUNFREED to display it.
 4   */


 7  #include "mpatrol.h"


10  int main(void)
11  {
12      void *p;
```

```
14        p = malloc(4);
15        p = malloc(4);
16        if (p != NULL)
17            free(p);
18        return EXIT_SUCCESS;
19  }
```

   The information that we are interested in comes after the summary of
library statistics generated in the log file. The first block of data
shows a memory map of the heap that is being handled by mpatrol. This
can be used to see graphically where a particular allocation is
located, or to look for memory fragmentation. The SHOWMAP option also
displays this information.

   Note that gaps in the memory map can either be due to space used by
internal memory blocks or to some other memory allocation library using
up space. On some systems that don't have virtual memory, gaps are
likely to be owned by other processes or belong to the system free
memory list.

```
    memory map:
      / 0x8000A000-0x8000A00F overflow (16 bytes)
      |+ 0x8000A010-0x8000A077 allocated (104 bytes) {malloc:1:0} [-|-|-]
      \ 0x8000A078-0x8000A087 overflow (16 bytes)
      / 0x8000A088-0x8000A097 overflow (16 bytes)
      |+ 0x8000A098-0x8000A115 freed (126 bytes) {free:2:0} [-|-|-]
      \ 0x8000A116-0x8000A125 overflow (16 bytes)
      / 0x8000A126-0x8000A135 overflow (16 bytes)
      |+ 0x8000A136-0x8000AF05 freed (3536 bytes) {free:3:0} [-|-|-]
      \ 0x8000AF06-0x8000AF15 overflow (16 bytes)
      / 0x8000AF16-0x8000AF25 overflow (16 bytes)
      |+ 0x8000AF26-0x8000AFA3 freed (126 bytes) {free:4:0} [-|-|-]
      \ 0x8000AFA4-0x8000AFB3 overflow (16 bytes)
      / 0x8000AFB4-0x8000AFC3 overflow (16 bytes)
      |+ 0x8000AFC4-0x8000AFC7 allocated (4 bytes) {malloc:10:0} [main|test.c|14]
      \ 0x8000AFC8-0x8000AFD7 overflow (16 bytes)
      / 0x8000AFD8-0x8000AFE7 overflow (16 bytes)
      |+ 0x8000AFE8-0x8000AFEB freed (4 bytes) {free:11:0} [main|test.c|17]
      \ 0x8000AFEC-0x8000AFFB overflow (16 bytes)
    --- 0x8000AFFC-0x8000AFFF free (4 bytes)
        ------------------- gap (12288 bytes)
      / 0x8000E000-0x8000E00F overflow (16 bytes)
      |+ 0x8000E010-0x8000EA27 freed (2584 bytes) {free:5:0} [-|-|-]
      \ 0x8000EA28-0x8000EA37 overflow (16 bytes)
      / 0x8000EA38-0x8000EA47 overflow (16 bytes)
      |+ 0x8000EA48-0x8000EAC5 freed (126 bytes) {free:6:0} [-|-|-]
      \ 0x8000EAC6-0x8000EAD5 overflow (16 bytes)
      / 0x8000EAD6-0x8000EAE5 overflow (16 bytes)
      |+ 0x8000EAE6-0x8000EB63 freed (126 bytes) {free:8:0} [-|-|-]
      \ 0x8000EB64-0x8000EB73 overflow (16 bytes)
    --- 0x8000EB74-0x8000EFFF free (1164 bytes)
        ------------------- gap (8192 bytes)
      / 0x80011000-0x8001100F overflow (16 bytes)
      |+ 0x80011010-0x800127F7 freed (6120 bytes) {free:7:0} [-|-|-]
      \ 0x800127F8-0x80012807 overflow (16 bytes)
    --- 0x80012808-0x80012FFF free (2040 bytes)
        ------------------- gap (106496 bytes)
```

```
       / 0x8002D000-0x8002D00F overflow (16 bytes)
     |+ 0x8002D010-0x8002DBBF freed (2992 bytes) {free:9:0} [-|-|-]
      \ 0x8002DBC0-0x8002DBCF overflow (16 bytes)
    --- 0x8002DBD0-0x8002DFFF free (1072 bytes)
```

   The next block of data shows a summary of all the symbols that could
be read from the program's executable file and/or any shared libraries
that the program requires. This can be useful to see which symbols
have actually been read by the mpatrol library. The SHOWSYMBOLS option
also displays this information.

   Note that the following data has been dramatically cut down in size
for the purposes of this example. The ... marks text that has been
removed.

```
    symbols read: 2438
        0x8000076C-0x800007D9 _init [./a.out] (110 bytes)
        0x80000900-0x8000094F _start [./a.out] (80 bytes)
        0x80000950-0x8000096F __do_global_dtors_aux [./a.out] (32 bytes)
        0x80000970-0x80000977 fini_dummy [./a.out] (8 bytes)
        ...
        0x80003B24-0x80003B4B __clear_cache [./a.out] (40 bytes)
        0x80003B4C-0x80003B6F __do_global_ctors_aux [./a.out] (36 bytes)
        0x80003B70-0x80003B77 init_dummy [./a.out] (8 bytes)
        0x80003B78-0x80003BA9 _fini [./a.out] (50 bytes)
        0xC0002604-0xC0002609 _start [/lib/ld.so.1] (6 bytes)
        0xC000260A-0xC0002659 _dl_start_user [/lib/ld.so.1] (80 bytes)
        0xC000265A-0xC0002B1B _dl_start [/lib/ld.so.1] (1218 bytes)
                  0xC000266A here [/lib/ld.so.1] (0 bytes)
        ...
        0xC0007A78-0xC0007AB5 __libc_read [/lib/ld.so.1] (62 bytes)
                  0xC0007A78 read [/lib/ld.so.1] (0 bytes)
                  0xC0007A9A __syscall_error [/lib/ld.so.1] (0 bytes)
        0xC0007AB8-0xC0007ADF __clear_cache [/lib/ld.so.1] (40 bytes)
        0xC0013E70-0xC0013E8B __mp_newlist [/usr/lib/libmpatrol.so.1.0] (28 bytes ←
            )
        0xC0013E8C-0xC0013EB3 __mp_addhead [/usr/lib/libmpatrol.so.1.0] (40 bytes ←
            )
        0xC0013EB4-0xC0013EE7 __mp_addtail [/usr/lib/libmpatrol.so.1.0] (52 bytes ←
            )
        0xC0013EE8-0xC0013F1B __mp_prepend [/usr/lib/libmpatrol.so.1.0] (52 bytes ←
            )
        ...
        0xC001A0DC-0xC001A0FF __nw__FUi [/usr/lib/libmpatrol.so.1.0] (36 bytes)
        0xC001A100-0xC001A123 __arr_nw__FUi [/usr/lib/libmpatrol.so.1.0] (36 ←
            bytes)
        0xC001A124-0xC001A143 __dl__FPv [/usr/lib/libmpatrol.so.1.0] (32 bytes)
        0xC001A144-0xC001A163 __arr_dl__FPv [/usr/lib/libmpatrol.so.1.0] (32 ←
            bytes)
        0xC003BB14-0xC003BB45 __libc_global_ctors [/lib/libc.so.6] (50 bytes)
        0xC003BB48-0xC003BB97 __libc_init [/lib/libc.so.6] (80 bytes)
        0xC003BB98-0xC003BBC3 __libc_print_version [/lib/libc.so.6] (44 bytes)
        0xC003BBC4-0xC003BBD7 __libc_main [/lib/libc.so.6] (20 bytes)
        ...
        0xC008F8BC-0xC008FA4D __moddi3 [/lib/libc.so.6] (402 bytes)
        0xC008FA50-0xC008FB19 __udivdi3 [/lib/libc.so.6] (202 bytes)
        0xC008FB1C-0xC008FC1B __umoddi3 [/lib/libc.so.6] (256 bytes)
```

```
        0xC008FC1C-0xC008FC4D _fini [/lib/libc.so.6] (50 bytes)
```

   The next block of data shows a summary of all freed memory
allocations. This is only possible because the NOFREE option was also
given, otherwise there would be no details on freed memory allocations.
All of these entries show where the allocation was freed, which can be
useful if you quickly needed to see where an allocation was freed. The
SHOWFREED option also displays this information.

   As this example was run on UNIX, the mpatrol library replaces the
default implementations of malloc(), free(), etc. As can be seen below,
this allows the library to trace all calls to allocate dynamic memory
in a process, even from functions that were not compiled with mpatrol.
The two functions shown below were called by the mpatrol library in
order to read the symbols from ELF object files. However, they are
located in the ELF access library which was not compiled with mpatrol.

   Note that the following data has again been cut down in size for the
purposes of this example. The ... marks text that has been removed.

```
    freed allocations: 9 (15740 bytes)
        0x8000A098 (126 bytes) {free:2:0} [-|-|-]
            0x800011BC elf_end
            0xC0019668 __mp_init
            0xC001982A __mp_alloc
            0x8000099C main
            0x80000944 _start

        0x8000A136 (3536 bytes) {free:3:0} [-|-|-]
            0x8000104E _elf_free
            0xC0019668 __mp_init
            0xC001982A __mp_alloc
            0x8000099C main
            0x80000944 _start

        ...
```

   The final block of data shows a summary of all unfreed memory
allocations. This can show up memory leaks, although the first unfreed
memory allocation in this example comes from the standard C library.
On systems such as UNIX it does not really matter about these unfreed
allocations since they will automatically be returned to the system on
process termination.

   However, the second unfreed allocation shows an example of a memory
leak, where no pointers referencing that allocation remain in the
program to free it with. If this was within a loop then the program
could quickly run away with memory, causing at least a decrease in
performance, and at most a memory shortage. The mpatrol library makes
it easier to spot memory leaks.

   The SHOWUNFREED option also displays this information.

```
    unfreed allocations: 2 (108 bytes)
        0x8000A010 (104 bytes) {malloc:1:0} [-|-|-]
            0xC0052B4A _IO_fopen
            0xC0017A0C __mp_openlogfile
```

```
        0xC0019648 __mp_init
        0xC001982A __mp_alloc
        0x8000099C main
        0x80000944 _start

    0x8000AFC4 (4 bytes) {malloc:10:0} [main|test.c|14]
        0x8000099C main
        0x80000944 _start
```

1.35 mpatrol.guide/Tutorial

```
Tutorial
********
```

   In this chapter we'll look at a real example of using the mpatrol
library to debug a program. All of the following building and
debugging steps were performed on a Linux/m68k machine so the details
may differ slightly on your system, but the concepts should remain the
same. However, on systems without virtual memory some of the steps may
actually cause the machine to lock up or crash so be aware of this if
you are running such a system -- you may be safer just reading this
tutorial rather than attempting it!

   This tutorial will also make use of the option USEDEBUG which
displays source-level file names and line numbers associated with
symbols in call stack tracebacks, but only if the underlying object
file access library supports reading line tables from object files and
even then only if the object files were compiled with debugging
information enabled.

   The program we are going to look at is a simple filter which
processes its standard input and displays the processed information on
its standard output. In this case the program converts all lowercase
characters to uppercase and removes any blank lines. The source for
the program is given below, but can also be found in
tests/tutorial/test1.c.

```
    23  /*
    24   * Reads the standard input file stream, converts all lowercase
    25   * characters to uppercase, and displays all non-empty lines to the
    26   * standard output file stream.
    27   */


    30  #include <stdio.h>
    31  #include <stdlib.h>
    32  #include <string.h>
    33  #include <ctype.h>


    36  char *strtoupper(char *s)
    37  {
    38      char *t;
    39      size_t i, l;
```

```
41          l = strlen(s);
42          t = (char *) malloc(l);
43          for (i = 0; i < l; i++)
44              t[i] = toupper(s[i]);
45          t[i] = '\0';
46          return t;
47      }


50      int main(void)
51      {
52          char *b, *s;

54          b = (char *) malloc(BUFSIZ);
55          while (gets(b))
56          {
57              s = strtoupper(b);
58              if (*s != '\0')
59              {
60                  puts(s);
61                  free(s);
62              }
63          }
64          free(b);
65          return EXIT_SUCCESS;
66      }
```

   If you quickly skimmed over the above code then you might have
noticed some rather obvious errors, but there are also some less
obvious ones hidden there as well. After compiling and linking with
the system C compiler and libraries it successfully runs, even when its
source code is piped to it. So if it runs, why bother trying to debug
it?

   The short answer to that is that this program does in fact contain
one rather major error that is likely to prevent it from running
portably on other systems. However, for the purposes of this tutorial,
we'll pretend that we've just been handed the source code for this
program and have not worked on it before. So let's now try to compile
and link it with the mpatrol library(1).

   First, add the inclusion of mpatrol.h to line 34 so that we can
replace calls to malloc() and free() with their mpatrol equivalents(2).
Then, recompile the program and link it with the mpatrol library.
This time, running it with even the simplest of non-empty input lines
should cause it to abort!

   If you look at the mpatrol.log file produced, you should see
something along the lines of the following at the end of the log file.

```
    ERROR: free memory corruption at 0x8000706C
            0x8000706C  00555555 55555555 55555555 55555555  .UUUUUUUUUUUUUUU
            0x8000707C  55555555 55555555 55555555 55555555  UUUUUUUUUUUUUUUU
            0x8000708C  55555555 55555555 55555555 55555555  UUUUUUUUUUUUUUUU
            0x8000709C  55555555 55555555 55555555 55555555  UUUUUUUUUUUUUUUU
```

```
        0x800070AC   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x800070BC   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x800070CC   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x800070DC   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x800070EC   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x800070FC   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x8000710C   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x8000711C   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x8000712C   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x8000713C   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x8000714C   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
        0x8000715C   55555555  55555555  55555555  55555555     UUUUUUUUUUUUUUUU
```

   This tells us that something has written a zero byte into free
memory at location 0x8000706C. Unfortunately, the library only caught
it at the next call to one of its functions so it had already happened
somewhere in between the last call and the current call. Turning on
the LOGALL option in the MPATROL_OPTIONS environment variable allows us
to see the last successful function call to the mpatrol library.

```
    ALLOC: malloc (50, 8192 bytes, 2 bytes) [main|test1.c|54]
            0x80000A30 main (/usr/users/homedir/graeme/test1.c:54)
            0x80000944 _start

    returns 0x80009000

    ALLOC: malloc (51, 4 bytes, 2 bytes) [strtoupper|test1.c|42]
            0x800009AE strtoupper (/usr/users/homedir/graeme/test1.c:42)
            0x80000A54 main (/usr/users/homedir/graeme/test1.c:57)
            0x80000944 _start

    returns 0x80007068
```

   Unfortunately, this only tells us that the last successful mpatrol
library function call was malloc() called from strtoupper(). If we add
the option OFLOWSIZE=8 to the MPATROL_OPTIONS environment variable then
we get slightly more information about which memory allocation was
affected(3).

```
    ERROR: allocation 0x80007080 has a corrupted overflow buffer at 0x80007084
            0x80007084   00AAAAAA  AAAAAAAA                        ........

      0x80007080 (4 bytes) {malloc:51:0} [strtoupper|test1.c|42]
            0x800009AE strtoupper (/usr/users/homedir/graeme/test1.c:42)
            0x80000A54 main (/usr/users/homedir/graeme/test1.c:57)
            0x80000944 _start
```

   Now we can make a better guess about what is happening. Since the
start of the upper overflow buffer of allocation 51 has been written
to, we can assume that something has written one byte beyond the end of
that memory allocation. You can probably see where that is happening
now by looking at the code, but let's try to be even more sure that
this is what is wrong.

   The only foolproof way to do this is to add a software watch point
to keep an eye on the address that is being written to. This can
normally only be done within a debugger, but on systems that support

programmable software watch points, the OFLOWWATCH option can be used
to do the same thing. For the sake of generality, we'll use the
debugger watch point approach, in this case with gdb. In order for the
following example to work correctly you'll need to add the ALLOCSTOP=51
option to the MPATROL_OPTIONS environment variable so that we can stop
just after the last successful memory allocation.

```
    (gdb) break main
    Breakpoint 1 at 0x80000a10: file test1.c, line 54.
    (gdb) run
    Starting program: a.out
    Breakpoint 1, main() at test1.c:54
    54          b = malloc(BUFSIZ);
    (gdb) break __mp_trap
    Breakpoint 2 at 0xc00182ac
    (gdb) continue
    Continuing.
    test
    Breakpoint 2, 0xc00182ac in __mp_trap()
    (gdb) backtrace
    #0  0xc00182ac in __mp_trap()
    #1  0xc0016494 in __mp_getmemory()
    #2  0xc001a618 in __mp_alloc()
    #3  0x800009ae in strtoupper(s=0x80009008 "test") at test1.c:42
    #4  0x80000a54 in main() at test1.c:57
    (gdb) step
    Single stepping until exit from function __mp_trap,
    which has no line number information.
    0xc0016494 in __mp_getmemory()
    (gdb) step
    Single stepping until exit from function __mp_getmemory,
    which has no line number information.
    0xc001a618 in __mp_alloc()
    (gdb) step
    Single stepping until exit from function __mp_alloc,
    which has no line number information.
    strtoupper(s=0x80009008 "test") at test1.c:43
    43          for (i = 0; i < l; i++)
    (gdb) watch *0x80007084
    Watchpoint 3: *2147512452
    (gdb) continue
    Continuing.
    Watchpoint 3: *2147512452
    Old value = -1431655766
    New value = 11184810
    strtoupper(s=0x80009008 "test") at test1.c:46
    46          return t;
    (gdb) quit
    The program is running.  Quit anyway (and kill it)? (y or n) y
```

After loading the program into gdb, we need to break at main() so
that we can run to a point where all of the shared library symbols have
been loaded into memory(4). We can then set another breakpoint at
__mp_trap() and continue until allocation 51 has been reached.

Because the mpatrol library has not been built with debugging
information in this example we can quickly step back to the

strtoupper() function since gdb won't step through functions that have
no debugging information. We then set a watch point on address
0x80007084, which is the address of the memory location that has been
causing the problems. After continuing, the debugger stops at line 46,
but this is more likely to be line 45 since that is where a zero byte
is being written to(5).

   So, we have located the problem, which is simply a case of not
allocating enough memory to contain the copied string and the
terminating zero byte. We can also improve the strtoupper() function
by checking the pointer returned by malloc() to see if it is NULL, and
if so simply exit with an error. You can try running the program with
the FAILFREQ option to see how it would originally behave in a low
memory situation.

   The following listing shows the above modifications that we have
made to our program. It can also be found in tests/tutorial/test2.c.

```
23  /*
24   * Reads the standard input file stream, converts all lowercase
25   * characters to uppercase, and displays all non-empty lines to the
26   * standard output file stream.
27   */


30  #include <stdio.h>
31  #include <stdlib.h>
32  #include <string.h>
33  #include <ctype.h>
34  #include "mpatrol.h"


37  char *strtoupper(char *s)
38  {
39      char *t;
40      size_t i, l;

42      l = strlen(s);
43      if ((t = (char *) malloc(l + 1)) == NULL)
44      {
45          fputs("strtoupper: out of memory\n", stderr);
46          exit(EXIT_FAILURE);
47      }
48      for (i = 0; i < l; i++)
49          t[i] = toupper(s[i]);
50      t[i] = '\0';
51      return t;
52  }


55  int main(void)
56  {
57      char *b, *s;

59      b = (char *) malloc(BUFSIZ);
60      while (gets(b))
61      {
```

```
62              s = strtoupper(b);
63              if (*s != '\0')
64              {
65                  puts(s);
66                  free(s);
67              }
68          }
69      free(b);
70      return EXIT_SUCCESS;
71  }
```

Leaving aside the obvious problem with gets() and the general
inefficiency of the algorithm, we could assume that our program works
safely now and we can release it to the outside world. However, a user
soon reports a problem with our program steadily using more and more
memory during its execution when processing very large files.

This is generally attributable to a memory leak and so we can use the
SHOWUNFREED option to try to detect where the memory leak is coming
from. Following is some example output from the mpatrol log file when
our program is run and is given a relatively small text file as input.

```
    unfreed allocations: 6 (109 bytes)
        0x80007000 (104 bytes) {malloc:1:0} [-|-|-]
            0xC008DB4A _IO_fopen
            0xC00183DC __mp_openlogfile
            0xC001A3A4 __mp_init
            0xC001A584 __mp_alloc
            0x80000A98 main
            0x80000980 _start

        0x80007068 (1 byte) {malloc:52:0} [strtoupper|test2.c|43]
            0x800009EE strtoupper
            0x80000ABC main
            0x80000980 _start

        0x8000706A (1 byte) {malloc:54:0} [strtoupper|test2.c|43]
            0x800009EE strtoupper
            0x80000ABC main
            0x80000980 _start

        0x8000706C (1 byte) {malloc:56:0} [strtoupper|test2.c|43]
            0x800009EE strtoupper
            0x80000ABC main
            0x80000980 _start

        0x8000706E (1 byte) {malloc:58:0} [strtoupper|test2.c|43]
            0x800009EE strtoupper
            0x80000ABC main
            0x80000980 _start

        0x80007070 (1 byte) {malloc:60:0} [strtoupper|test2.c|43]
            0x800009EE strtoupper
            0x80000ABC main
            0x80000980 _start
```

We can discount the first entry since that is obviously coming from

when the mpatrol library first initialises itself. However, all of the
other entries appear to be coming from line 43 within strtoupper() and
appear to be only 1 byte in length. At that point in the code, the
only possible reason for allocating 1 byte is when the string is empty
and so that must mean that we are not freeing memory that contains
empty strings. Looking at line 66 we can see that free() is only ever
called for non-empty strings and therefore if we move the call to
free() outside the test for an empty string we will fix the memory
leak. The file tests/tutorial/test3.c contains the source for the
final program.

    ---------- Footnotes ----------

    (1) On UNIX systems with dynamic linking it might also be possible
to run the program under the mpatrol command with its -d option without
having to recompile or relink, but compiling and linking with the
mpatrol library is a more generic solution across different platforms.

    (2) This is not strictly necessary on UNIX and Windows platforms
(and AmigaOS when using gcc), but it does give us more debugging
information.

    (3) Note that the start address of the allocation has changed
slightly since we added padding around it with the OFLOWSIZE option.

    (4) This is only necessary when the mpatrol library has been built
as a shared library.

    (5) This is not necessarily the fault of the debugger or the
debugging information generated by the compiler since on most platforms
such watchpoints can only be caught after they occur, hence most
debuggers show the next statement to be executed rather than the current
one.

1.36   mpatrol.guide/Functions

Functions
****

   The mpatrol library contains implementations of dynamic memory
allocation functions for C and C++ suitable for tracing and debugging.
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. It is
intended that the NDEBUG preprocessor macro be defined in release
builds.

The following 14 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.

void *malloc(size_t size)
    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().

void *calloc(size_t nelem, size_t size)
    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().

void *memalign(size_t align, size_t size)
    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.

void *valloc(size_t size)
    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.

void *pvalloc(size_t size)
    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.

char *strdup(const char *str)
    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 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().

char *strndup(const char *str, size_t size)
    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 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.

char *strsave(const char *str)
    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 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.

char *strnsave(const char *str, size_t size)
    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 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.

void *realloc(void *ptr, size_t size)
    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().

void *recalloc(void *ptr, size_t nelem, size_t size)
    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.

void *expand(void *ptr, size_t size)
    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.

void free(void *ptr)
    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.

void cfree(void *ptr, size_t nelem, size_t size)
    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.

   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. 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.

void *operator new(size_t size)
    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 -- no exceptions will
    be thrown. The allocated memory must be deallocated with operator
    delete.

void *operator new[](size_t size)
    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 -- no exceptions will
    be thrown. The allocated memory must be deallocated with operator
    delete[].

void operator delete(void *ptr)
    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.

void operator delete[](void *ptr)
     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[].

void (*set_new_handler(void (*func)(void)))(void)
     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. The
     default low-memory handler for the C++ operators will terminate
     the program and write an out of memory message to the log file.
     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.

void *memset(void *ptr, int byte, size_t size)
     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.

void bzero(void *ptr, size_t size)
     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.

void *memccpy(void *dest, const void *src, int byte, size_t size)
     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.

void *memcpy(void *dest, const void *src, size_t size)
     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.

void *memmove(void *dest, const void *src, size_t size)
    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.

void bcopy(const void *src, void *dest, size_t size)
    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.

int memcmp(const void *ptr1, const void *ptr2, size_t size)
    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.

int bcmp(const void *ptr1, const void *ptr2, size_t size)
    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.

void *memchr(const void *ptr, int byte, size_t size)
    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.

void *memmem(const void *ptr1, size_t size1, const void *ptr2, size_t size2)
    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 8 functions are available as support routines for
additional control and tracing in the mpatrol library. To use these
you should include the mpatrol.h header file.

int __mp_info(const void *ptr, __mp_allocinfo *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 then 0 will be returned,
     otherwise 1 will be returned and INFO will contain the following
     information:

     Field   Description
     block   Pointer to first byte of allocation.
     size   Size of allocation in bytes.
     type   Type of function which allocated memory.
     alloc   Allocation index.
     realloc   Number of times reallocated.
     thread   Thread identifier.
     func   Function in which allocation took place.
     file   File in which allocation took place.
     line   Line number at which allocation took place.
     stack   Pointer to function call stack.
     freed   Indicates if allocation has been freed.

int __mp_printinfo(const void *ptr)
     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 then 0 will be returned,
     otherwise 1 will be returned. This function is intended to be
     called from within a debugger.

void __mp_memorymap(int stats)
     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.

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

void __mp_check(void)
     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.

void (*__mp_prologue(void (*func)(const void *, size_t)))(const void *, size_t)
     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:

```
        Argument 1      Argument 2      Called by
        -1              SIZE            malloc(), etc.
        PTR             SIZE            realloc(), etc.
        PTR             -1              free(), etc.
        PTR             -2              strdup(), etc.
```

void (*__mp_epilogue(void (*func)(const void *)))(const void *)
     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:

```
        Argument        Called by
        PTR             malloc(), realloc(), strdup(), etc.
        -1              free(), etc.
```

void (*__mp_nomemory(void (*func)(void)))(void)
     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. Note
     that this function is equivalent to set_new_handler() and will
     replace the handler installed by that function.

1.37   mpatrol.guide/Environment

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. 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. 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=-.

CHECKALL
    Equivalent to the CHECKALLOCS, CHECKREALLOCS and CHECKFREES
    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.

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

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.

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.

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.

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. 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.

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
     Specifies that the mpatrol library should keep all reallocated and
     freed memory allocations. Such freed memory allocations will then
     be flagged as freed and can be used by the library to provide
     better diagnostics. However, as no system memory will ever be
     reused by the mpatrol library, this option can quickly lead to a
     shortage of available system memory for a process. Note that this
     option 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.

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. 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 be
sent to the stderr file stream. Default value:
PROFFILE=mpatrol.out.

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 SHOWFREED, SHOWUNFREED, SHOWMAP and SHOWSYMBOLS
    options specified together.

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.

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.

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. 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.
Note that this option will slow down program execution, use up
more system memory and may leave unaccounted unfreed memory
allocations at program termination.

USEMMAP
Specifies that the library should use mmap() instead of sbrk() to
allocate system 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. It is ignored on systems that do not support
the mmap() system call.

1.38 mpatrol.guide/Options

                Options
****

   A utility program called mpatrol is provided to run commands that
have been linked with the mpatrol library.

     mpatrol [options] <command> [arguments]

   The mpatrol command is used to set various mpatrol library OPTIONS
when running COMMAND with its ARGUMENTS. In most cases, COMMAND must
have been linked with the mpatrol library, unless the -d option is used
in which case COMMAND need only have been dynamically linked.

   All mpatrol library diagnostics are sent to the file mpatrol.%n.log
in the current directory by default (where %n is the current process
id) but this can be changed using the -l option. Similarly, the default
profiling output filename is mpatrol.%n.out. Note that the LOGALL
option is always implicitly used for commands that are run by this
command.

   Alternatively, the log file and profiling output file names can
contain %p, which will be replaced with the name of the program being
executed without the directory components. If the executable filename
could not be determined or was not set then it will be replaced with
mpatrol.

   All of the following options (except -d and -V) correspond to their
listed mpatrol library option (see
                Environment
).

-1 <UNSIGNED-INTEGER>
    [SMALLBOUND] Specifies the limit in bytes up to which memory
    allocations should be classified as small allocations for
    profiling purposes.

-2 <UNSIGNED-INTEGER>
    [MEDIUMBOUND] Specifies the limit in bytes up to which memory
    allocations should be classified as medium allocations for
    profiling purposes.

-3 <UNSIGNED-INTEGER>
    [LARGEBOUND] Specifies the limit in bytes up to which memory
    allocations should be classified as large allocations for
    profiling purposes.

-A <UNSIGNED-INTEGER>
    [ALLOCSTOP] Specifies an allocation index at which to stop the
    program when it is being allocated.

-a <UNSIGNED-INTEGER>
    [ALLOCBYTE] Specifies an 8-bit byte pattern with which to prefill
    newly-allocated memory.

-C <UNSIGNED-RANGE>
    [CHECK] Specifies a range of allocation indices at which to check
    the integrity of free memory and overflow buffers.

-c
    [CHECKALL] Specifies that all arguments to functions which
    allocate, reallocate and deallocate memory have rigorous checks
    performed on them.

-D <UNSIGNED-INTEGER>
    [DEFALIGN] Specifies the default alignment for general-purpose
    memory allocations, which must be a power of two.

-d
    Specifies that programs which were not linked with the mpatrol
    library should also be traced, but only if they were dynamically

        linked. This option will only work if the system dynamic linker
        has the ability to preload a set of user-specified shared
        libraries via a special environment variable.

-e <STRING>
        [PROGFILE] Specifies an alternative filename with which to locate
        the executable file containing the program's symbols.

-F <UNSIGNED-INTEGER>
        [FREESTOP] Specifies an allocation index at which to stop the
        program when it is being freed.

-f <UNSIGNED-INTEGER>
        [FREEBYTE] Specifies an 8-bit byte pattern with which to prefill
        newly-freed memory.

-G
        [SAFESIGNALS] Instructs the library to save and replace certain
        signal handlers during the execution of library code and to
        restore them afterwards.

-g
        [USEDEBUG] Specifies that any debugging information in the
        executable file should be used to obtain additional source-level
        information.

-L <UNSIGNED-INTEGER>
        [LIMIT] Specifies the limit in bytes at which all memory
        allocations should fail if the total allocated memory should
        increase beyond this.

-l <STRING>
        [LOGFILE] Specifies an alternative file in which to place all
        diagnostics from the mpatrol library.

-M
        [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.

-m
        [USEMMAP] Specifies that the library should use mmap() instead of
        sbrk() to allocate system memory.

-N
        [NOPROTECT] Specifies that the mpatrol library's internal data
        structures should not be made read-only after every memory
        allocation, reallocation or deallocation.

-n
        [NOFREE] Specifies that the mpatrol library should keep all
        reallocated and freed memory allocations.

-O <UNSIGNED-INTEGER>
        [OFLOWSIZE] Specifies the size in bytes to use for all overflow
        buffers, which must be a power of two.

-o <UNSIGNED-INTEGER>
     [OFLOWBYTE] Specifies an 8-bit byte pattern with which to fill the
     overflow buffers of all memory allocations.

-P <STRING>
     [PROFFILE] Specifies an alternative file in which to place all
     memory allocation profiling information from the mpatrol library.

-p
     [PROF] Specifies that all memory allocations are to be profiled
     and sent to the profiling output file.

-Q <UNSIGNED-INTEGER>
     [AUTOSAVE] Specifies the frequency at which to periodically write
     the profiling data to the profiling output file.

-R <UNSIGNED-INTEGER>
     [REALLOCSTOP] Specifies an allocation index at which to stop the
     program when a memory allocation is being reallocated.

-S
     [SHOWMAP & SHOWSYMBOLS] Specifies that a memory map of the entire
     heap and a summary of all of the function symbols read from the
     program's executable file should be displayed at the end of
     program execution.

-s
     [SHOWFREED & SHOWUNFREED] Specifies that a summary of all of the
     freed and unfreed memory allocations should be displayed at the
     end of program execution.

-U <UNSIGNED-INTEGER>
     [UNFREEDABORT] Specifies the minimum number of unfreed
     allocations at which to abort the program just before program
     termination.

-V
     Displays the version number of the mpatrol command.

-v
     [PRESERVE] Specifies that any reallocated or freed memory
     allocations should preserve their original contents.

-w
     [OFLOWWATCH] Specifies that watch point areas should be used for
     overflow buffers rather than filling with the overflow byte.

-X
     [PAGEALLOC=UPPER] Specifies that each individual memory allocation
     should occupy at least one page of virtual memory and should be
     placed at the highest point within these pages.

-x
     [PAGEALLOC=LOWER] Specifies that each individual memory allocation
     should occupy at least one page of virtual memory and should be
     placed at the lowest point within these pages.

```
-Z <UNSIGNED-INTEGER>
     [FAILSEED]  Specifies the random number seed which will be used
     when determining which memory allocations will randomly fail.

-z <UNSIGNED-INTEGER>
     [FAILFREQ]  Specifies the frequency at which all memory allocations
     will randomly fail.
```

1.39 mpatrol.guide/Library performance

```
Library performance
*******************

     The following times were obtained on a Sun Ultra 5 with an
UltraSPARC IIi processor running at 333MHz and running Solaris 7.  The
test performed was the one in tests/pass/test1.c and all tests were run
on a lightly loaded system, but were run several times to obtain an
average result.  Obviously, these times can only be an approximation,
but should serve to illustrate the effects on performance that each
option can have.  All times are given in seconds, and the second time
on each line was obtained with the same options plus the NOPROTECT
option.  Running with the CHECK=0 option would speed things up
dramatically, albeit at the expense of less error checking.

     Running with basic options:

no options                                    0.618        0.258
OFLOWSIZE=2                                    0.645        0.296
OFLOWSIZE=8                                    0.686        0.327
PAGEALLOC=LOWER                               7.785        7.372
PAGEALLOC=UPPER                               7.821        7.469

     Running when all freed memory allocations are kept:

NOFREE                                        0.943        0.506
NOFREE OFLOWSIZE=2                            1.026        0.579
NOFREE OFLOWSIZE=8                            1.091        0.645
NOFREE PAGEALLOC=LOWER                        8.013        7.598
NOFREE PAGEALLOC=UPPER                        8.026        7.616

     Running when all freed memory allocations are kept and their
contents are preserved:

NOFREE PRESERVE                               0.719        0.292
NOFREE PRESERVE OFLOWSIZE=2                   0.792        0.367
NOFREE PRESERVE OFLOWSIZE=8                   0.850        0.419
NOFREE PRESERVE PAGEALLOC=LOWER               8.043        7.616
NOFREE PRESERVE PAGEALLOC=UPPER               8.052        7.631

     Running using watch points to check the overflow buffers:

OFLOWSIZE=2 OFLOWWATCH                         Interrupted after half an
```

```
                                                              hour as it still hadn't
                                                              finished.

    Running using the Solaris 7 malloc libraries:

Solaris 7 malloc(3c) library                      0.033
Solaris 7 malloc(3x) library                      0.036
Solaris 7 bsdmalloc(3x) library                   0.028
Solaris 7 mapmalloc(3x) library                   0.033
Solaris 7 watchmalloc(3x) library                 40.845
```

1.40   mpatrol.guide/Profiling file format

```
Profiling file format
*********************
```

   The format of the profiling output files that are produced by the
mpatrol library is described here. Every profiling output file
contains the following components.

   * 4 bytes containing the characters M, P, T and L.

   * 1 unsigned integer representing the value 1. This is used by
     mprof to determine the endianness of the processor that produced
     the profiling output file so that it can decide whether to perform
     byte-swapping on the input data.

   * 3 unsigned integers containing the small, medium and large
     allocation bounds.

   * 1 unsigned integer containing the allocation bin size. If the
     allocation bin size is greater than zero then it is followed by
     the allocation bins, the large allocation totals, the deallocation
     bins and the large deallocation totals, where the bins are arrays
     of unsigned integers with dimensions of the allocation bin size
     and the totals are unsigned integers.

   * 1 unsigned integer containing the number of profiling data
     structures. If the number of profiling data structures is greater
     than zero then it is followed by the profiling data structures
     themselves, which are of the following structure.

       * 1 unsigned integer representing the index of this profiling
         data.

       * 4 unsigned integers representing the small, medium, large and
         extra large allocation counts for this profiling data.

       * 4 unsigned integers representing the small, medium, large and
         extra large allocation totals for this profiling data.

       * 4 unsigned integers representing the small, medium, large and
         extra large deallocation counts for this profiling data.

               * 4 unsigned integers representing the small, medium, large and
                 extra large deallocation totals for this profiling data.

     * 1 unsigned integer containing the number of call sites. If the
       number of call sites is greater than zero then it is followed by
       the call sites themselves, which are of the following structure.

            * 1 unsigned integer representing the index of this call site.

            * 1 unsigned integer representing the index of the parent call
              site.

            * 1 generic pointer representing the code address of this call
              site.

            * 1 unsigned integer representing the index of an associated
              symbol.

            * 1 unsigned integer representing the offset of the symbol name.

            * 1 unsigned integer representing the index of any associated
              profiling data.

     * 1 unsigned integer containing the number of symbol addresses. If
       the number of symbol addresses is greater than zero then it is
       followed by the symbol addresses themselves, which are generic
       pointers.

     * 1 unsigned integer containing the size of the symbol name string
       table. This is followed by the symbol name string table, which is
       an array of characters containing the nul-terminated symbol names.

     * 4 bytes containing the characters M, P, T and L.

1.41 mpatrol.guide/Supported systems

                  Supported systems
****

    Following is a list of systems on which the mpatrol library has been
built and tested. The system details include the operating system and
version, the processor type, the object file format and the C compiler
used to compile the library and tests. The details following each
system list any features of the library that are not (or cannot be)
supported on that system.

     * AIX 4.1, IBM RS/6000, BFD, cc
            * The thread-safe version of the library does not work.

            * The OFLOWWATCH option has no effect.

            * Cannot automatically determine the program filename.

            * No support for call stack traversal.

        * The address of an illegal memory access cannot be determined.

        * The -d option to the mpatrol command has no effect.

  * DG/UX 4.20MU07, Intel Pentium Pro, ELF32, gcc
        * The thread-safe version of the library does not work.

        * The OFLOWWATCH option has no effect.

        * The USEDEBUG option has no effect.

        * The -d option to the mpatrol command does not work unless
          libelf.so is available.

  * DG/UX 4.11, Intel Pentium Pro, ELF32, gcc
        * The thread-safe version of the library does not work.

        * The OFLOWWATCH option has no effect.

        * The USEDEBUG option has no effect.

        * The -d option to the mpatrol command has no effect.

  * DG/UX 4.11, Motorola 88100, ELF32, gcc
        * The thread-safe version of the library does not work.

        * The OFLOWWATCH option has no effect.

        * The USEDEBUG option has no effect.

        * Cannot automatically determine the program filename.

        * Call stack traversal only works with unoptimised code.

        * The -d option to the mpatrol command has no effect.

  * DYNIX/ptx 4.5, Intel Pentium Pro, ELF32, cc
        * The thread-safe version of the library does not work.

        * The OFLOWWATCH option has no effect.

        * The USEDEBUG option has no effect.

        * The -d option to the mpatrol command has no effect.

  * HP/UX 10.20, HP PA/RISC 9000, BFD, gcc
        * The thread-safe version of the library does not work.

        * The OFLOWWATCH option has no effect.

        * The USEMMAP option has no effect.

        * Cannot automatically determine the program filename.

        * No support for call stack traversal.

* The -d option to the mpatrol command has no effect.

* IRIX 5.3, MIPS R4000, ELF32, cc
    * The thread-safe version of the library does not work.

    * The OFLOWWATCH option has no effect.

    * The USEDEBUG option has no effect.

    * No support for call stack traversal.

    * Symbols cannot be read from any shared libraries that a
      program uses.

* RedHat Linux 6.0, Intel Pentium III, BFD, gcc
    * The thread-safe version of the library does not work.

    * The OFLOWWATCH option has no effect.

    * The address of an illegal memory access cannot be determined.

    * The -d option to the mpatrol command does not work unless
      libiberty.so is available.

* RedHat Linux 5.1, Motorola 68040, BFD, gcc
    * The thread-safe version of the library does not work.

    * The OFLOWWATCH option has no effect.

    * The address of an illegal memory access cannot be determined.

    * The -d option to the mpatrol command does not work unless
      libiberty.so is available.

* RedHat Linux 5.1, Motorola 68040, ELF32, gcc
    * The thread-safe version of the library does not work.

    * The OFLOWWATCH option has no effect.

    * The USEDEBUG option has no effect.

    * The address of an illegal memory access cannot be determined.

    * The -d option to the mpatrol command does not work unless
      libelf.so is available.

* LynxOS 3.0.0, PowerPC, BFD, gcc
    * The thread-safe version of the library does not work.

    * The OFLOWWATCH option has no effect.

    * The USEMMAP option has no effect.

    * Cannot automatically determine the program filename.

    * No support for call stack traversal.

* The address of an illegal memory access cannot be determined.

* The -d option to the mpatrol command has no effect.

* Solaris 2.6, Intel Pentium Pro, BFD, gcc
    * The thread-safe version of the library does not work.

* Solaris 2.6, Intel Pentium Pro, ELF32, gcc
    * The thread-safe version of the library does not work.

    * The USEDEBUG option has no effect.

* Solaris 7, SPARC V9, BFD, gcc
    * The thread-safe version of the library does not work.

* Solaris 7, SPARC V9, ELF32, gcc
    * The thread-safe version of the library does not work.

    * The USEDEBUG option has no effect.

* AmigaOS 3.1, Motorola 68040, BFD, gcc
    * No memory protection so the PAGEALLOC option has no effect.

    * The OFLOWWATCH option has no effect.

    * The USEDEBUG option has no effect.

    * The USEMMAP option has no effect.

    * Limited support for call stack traversal.

    * Limited support for reading symbols.

    * No detection of illegal memory accesses.

    * The -d option to the mpatrol command has no effect.

* AmigaOS 3.1, Motorola 68040, n/a, SAS/C
    * No automatic override of malloc(), etc., without inclusion of
      mpatrol.h.

    * No memory protection so the PAGEALLOC option has no effect.

    * The OFLOWWATCH option has no effect.

    * The USEDEBUG option has no effect.

    * The USEMMAP option has no effect.

    * No support for call stack traversal.

    * No support for reading symbols.

    * No detection of illegal memory accesses.

    * The -d option to the mpatrol command has no effect.

* Microsoft Windows NT 4.0, Intel Pentium III, n/a, Microsoft Visual
  C/C++
    * The OFLOWWATCH option has no effect.

    * The USEDEBUG option has no effect.

    * The USEMMAP option has no effect.

    * No support for reading symbols.

    * The address of an illegal memory access cannot be determined.

    * The -d option to the mpatrol command has no effect.

                 New system support
                  Adding a new operating system.

                 New processor support
                  Adding a new processor architecture.

                 New file format support
                  Adding a new object file format.

1.42   mpatrol.guide/New system support

```
Adding a new operating system
=============================
```

* Add a new TARGET and/or SYSTEM definition in target.h. The TARGET
  macro is for fundamentally different operating systems, whereas
  the SYSTEM macro is for differentiating variations of a particular
  operating system.

* Make any necessary modifications to config.h.

* Add any support for memory allocation in memory.c.

* Add any support for stack traversal in stack.c.

* Add any support for signals in signals.c.

* Add any support for threads in mutex.c.

* Add any support for filenames in diag.c.

* Add a new version and date format (or use an existing one) in
  version.c.

* Decide if the malloc() replacements should be used from malloc.c.

* Add any support for invoking commands in mpatrol.c.

* Add a new subdirectory in the build directory that contains a
  Makefile and any other files that are required to build the
  library on the new operating system.

1.43   mpatrol.guide/New processor support

```
Adding a new processor architecture
===================================
```

* Add a new ARCH definition in target.h.

* Make any necessary modifications to config.h.

* Add any support for memory allocation in memory.c.

* Add any support for stack traversal in stack.c.

1.44   mpatrol.guide/New file format support

```
Adding a new object file format
===============================
```

* Add a new FORMAT definition in target.h.

* Make any necessary modifications to config.h.

* Add any support for stack traversal in stack.c.

* Add any support for symbol reading in symbol.c.

1.45   mpatrol.guide/Notes

```
              Notes
*****
```

   This section contains information about known bugs and limitations
in the mpatrol library as well as listing potential future enhancements.

   Bugs should be reported to <mpatrol@cbmamiga.demon.co.uk> along with
the details of the operating system, processor architecture and object
file format that the mpatrol library is being used with -- and don't
forget to include the version of the mpatrol library you are using!
Keep in mind that I only have access to an Amiga running RedHat
Linux/m68k 5.1 and AmigaOS 3.1, so I will be most likely unable to
reproduce most of the system-specific bugs. A bug report that comes
with an associated fix will be most welcome.

   Enhancement requests and source code containing enhancements should
also be sent to <mpatrol@cbmamiga.demon.co.uk> or the mpatrol
discussion group at 'http://www.egroups.com/group/mpatrol/'. If you
are planning to implement an enhancement, let me know first in case I
am (or someone else is) working towards the same goal -- that way, work
won't be wasted. If you wish to send me source code changes please
send the changes as context diffs or in an e-mail attachment as a
compressed tar archive.

                     Generic notes
                      Notes for all platforms.

                     UNIX notes
                      Notes for UNIX platforms.

                     Amiga notes
                      Notes for Amiga platforms.

                     Windows notes
                      Notes for Windows platforms.

                     Netware notes
                      Notes for Netware platforms.

1.46   mpatrol.guide/Generic notes

Notes for all platforms
=======================

   * C++ support is still fairly limited, and will possibly only work
     for older C++ code due to the way the operators are overridden
     (i.e. there are no exceptions versions of the functions). There
     are also likely to be potential problems with the macros which
     redefine malloc() and operator new, etc., since there may be
     member functions in code that will mistakenly be redefined if their
     names match the macro definitions, and also means that calls to
     placement new will not work at all. Also, explicit references to
     operator new rather than new are likely to result in compilation
     errors, and the way that source level information is obtained for
     operator delete means that the resulting code will not be
     thread-safe.

   * Need to add support for 64-bit processors. This shouldn't be too
     hard, but I haven't got access to a 64-bit environment to test it,
     so I haven't bothered yet.

   * The thread-safe code in the library doesn't yet work properly,
     probably because of the recursion flag which is incremented or
     decremented before the mutex is locked. Hence, the threads test
     (tests/pass/test5.c) doesn't work yet.

* Need to make the library re-entrant. This could be achieved by
  moving the static variables in memory.c, stack.c, mutex.c, diag.c,
  option.c and sbrk.c into the infohead structure and then having an
  array of infohead structures from which to allocate new memory
  headers when a new one is required. This is only necessary for
  Amiga shared libraries and Netware NLMs since UNIX and Windows
  platforms allocate a new copy of the data section in a shared
  library or DLL when it is opened by a new process.

* The current implementation of call stack traversal is limited and
  will only likely work for unoptimised code. A much better
  solution would be write the implementation at a lower level in
  assembly, but this is much less portable. Perhaps there is a
  library which can be used to perform this across many operating
  systems and processor architectures, or maybe someone would like to
  write one? I can think of many applications that would benefit
  from such a library besides this one(1).

* An alternative implementation for call stack traversal uses the
  functions __builtin_frame_address() and __builtin_return_address()
  that are available when the library is compiled with gcc.
  However, they can only traverse a number of stack frames at
  compile-time, not run-time so there is a maximum number of stack
  frames that can be traversed at any one time. The implementation
  depends on both of these builtin functions returning NULL when the
  top of stack is reached. If this is not the case then this method
  cannot be used or should only be used with a small number of fixed
  stack frames.

* Add a function to display a stack trace of the current call stack
  to a file stream and another function to walk all current memory
  allocations invoking a call-back function for each one
  encountered. Note that there is still an issue with call-back
  functions if they call mpatrol library functions, since this will
  lead to recursion.

* In object file formats that support nested symbols (such as ELF),
  the current implementation will tend to show some shortcomings.
  This is because there is currently no nesting count in the
  function that deals with symbol name lookup, so the wrong symbol
  name may be displayed in diagnostics.

* In object file formats that don't store the sizes of symbols (such
  as basic COFF, or when using the GNU BFD library), the current
  implementation will simply assume that the current symbol
  terminates at the beginning of the next symbol in the virtual
  address space.

* There still appears to be a problem with the code address to line
  number mapping provided by the GNU BFD library, and this needs to
  be looked into further. Also need to store filename and line
  number information in all call stacks so that the information can
  be used at program termination. May also need to display this
  information in the __mp_printinfo() function and add this
  information to the profiling output file so that mprof can make
  use of it.

* Perhaps add the ability to profile memory operations such as
  memcpy() and memset() to the existing memory allocation profiling
  facility. Also, add options to mprof to write out files that can
  be used by graph drawing software for a better visualisation of
  the profiling information. Finally, perhaps add an allocation
  call graph table to mprof, similar to that produced by gprof for
  execution call graphs.

* Perhaps add a memory usage profiling feature which would dump out
  memory usage statistics to a file at set intervals during run-time
  so that a table of memory usage could be built up. This could
  also be extended to having a graphical tool which could display
  this information.

* Add a CRC checksum to memory blocks and use it to check that freed
  memory allocations have not been corrupted when the NOFREE and
  PRESERVE options are in use on platforms which have no memory
  protection. This could also be extended to marking allocated
  memory blocks and then displaying what blocks have changed after a
  certain period from within a debugger. Another idea could be to
  display all memory allocations, etc. made since a certain
  function was called from within a program.

* Improve use of watch points by allowing an option which will only
  install write watch points instead of both read and write watch
  points. Not only will this speed up the use of watch points, but
  will also cause less problems with reading from misaligned memory
  allocations.

* Perhaps add memory protection to the simulated sbrk heap.

* Add a SHOWFREE option to display a list of all free memory blocks
  at program termination for debugging purposes to view memory
  fragmentation. If that option is added then perhaps SHOWALL
  should only be equivalent to SHOWFREE, SHOWFREED and SHOWUNFREED,
  and SHOWMAP and SHOWSYMBOLS should be explicitly given.

* Add an option, similar to NOFREE, that would prevent a freed memory
  allocation from being used until a certain number of memory
  allocations later. This would be far less of a resource-hogger
  than the NOFREE option and might catch just as many errors but
  might be extremely hard to implement.

* Add an extra piece of information in the log file summary which
  highlights the peak number of memory allocations in use at any one
  time during program execution.

* Add versions of mallopt(), mallinfo(), memorymap(), mallocctl(),
  mallocblksize() and msize() which are provided in many other
  malloc libraries. These won't necessarily behave in exactly the
  same way as existing implementations, but at least there won't be
  link errors when compiling source code which uses them.

* Perhaps add debugging/tracing versions of the string manipulation
  functions, such as strlen() and strcmp() in much the same way as
  was done for the memory operation functions. The only problem

with this would be locale support, but perhaps it might be easier
just to assume the C locale to begin with. Also need to have
better detection of internal and free blocks when displaying
memory range errors.

* Perhaps add definitions of xmalloc(), xrealloc(), etc. which never
  return NULL on failure, and perhaps also add definitions of
  XtMalloc(), XtRealloc(), etc. for X-Window programming. Some
  other malloc libraries provide versions of these but perhaps they
  are not needed if they are implemented using malloc(), realloc(),
  etc.

* Add another library which can be linked in instead of mpatrol and
  replaces all calls to __mp_alloc(), etc., with the original calls
  to malloc() and related functions. This would be very useful for
  quickly removing all mpatrol functionality for perhaps even a
  release build, and might be useful for implementing functions such
  as memalign() which don't exist on many systems.

* Write a set of functions that are compatible with those
  implemented by Checker, the gcc run-time memory access checker.
  This would allow every memory access to be checked in object files
  compiled with gcc, not just pointers into the heap, and would
  provide error checking as effective as source code
  instrumentation. Could also make use of the etext, edata and end
  pointers that are set at run-time on most UNIX systems.

* Perhaps remove the automatic logging of memory operations from
  programs run through the mpatrol command, and instead add an
  option to do it explicitly.

* Add an option to specify that all failed memory allocations should
  abort (or at least give a warning) instead of returning a NULL
  pointer. Also, perhaps add an option to display the partial
  contents of freed and unfreed allocations in the mpatrol log file.

* Perhaps use GNU autoconf to automatically work out values for
  config.h on the platform it is being built on, and also use
  automake, libtool and install when building and installing files.

* The postscript version of the quick reference card seems to print
  at an unusual offset on some printers. Also, the PDF version gets
  created with 1 inch margins rather than half inch margins which
  means that it is one inch too large in both dimensions. Need to
  figure out what is happening in both cases.

---------- Footnotes ----------

   (1) The latest release of the GNU C library includes a backtrace()
function which fills in an array of return addresses, but this requires
the presence of the library and some features of GCC.

1.47   mpatrol.guide/UNIX notes

Notes for UNIX platforms
========================

* Need to add watch point area support for non-Solaris operating
  systems. This may be a case of preventing all heap memory from
  being accessed and providing a signal handler that is called when
  a read from or write to such memory triggers a signal. The
  handler could then determine if the address is in a watch point,
  and if it is not it could unprotect the memory and return.

* Need to improve watch point facility in order to speed it up by an
  order of magnitudes. This will most likely involve removing all
  watch points when entering the library and replacing them when
  returning to user code.

* Need to add advanced signal information for operating systems that
  do not support the siginfo() system call. This information is
  used by the signal handler that handles the SIGSEGV signal in
  order to provide useful information about where an illegal memory
  access occurred. However, there is currently a problem in that
  the call stack displayed from within that handler is not
  necessarily accurate with respect to the function at the top of
  the stack. Also, signal handlers shouldn't technically call I/O
  functions in case of additional signals being caught so this may
  need to be improved.

* Need to add a portable way of initialising the thread-safe version
  of the library when it is compiled by a C compiler. There is
  already a solution to this problem when it is compiled by a C++
  compiler, but it may only work if the final program is linked with
  that specific compiler.

* Need to add support for call stack traversal for at least the
  Alpha, MIPS, PA/RISC, PowerPC and RS/6000 processor architectures.
  The current implementation of call stack traversal for the
  Motorola 88xx0 family is also a bit flaky and so should only be
  used when the library and program are built unoptimised.

* Need to add support for obtaining the program name from the stack
  for at least the Alpha, Motorola 88xx0, non-IRIX MIPS, PA/RISC,
  PowerPC and RS/6000 processor architectures. Also need to add
  support for reading the program symbols from a suitable file in
  /proc for other operating systems that support it. If there is no
  support for either of these methods then the PROGFILE option can
  currently be used to specify the program name at run-time.

* The library cannot currently read any symbols from shared objects
  that have been read via dlopen(). Also, on IRIX platforms no
  symbols can currently be read from any shared libraries that were
  used by a program. This is because SGI have a slightly different
  interface to their dynamic linker that I haven't been able to
  figure out yet.

* There is a problem on later Linux releases where the _DYNAMIC
  symbol is defined in elf.h, thus resulting in a conflicting
  definition when compiling symbol.c.

* The -d option to the mpatrol command does not always work on
  systems whose dynamic linkers support the LD_PRELOAD or _RLD_LIST
  environment variables. This is because the object file format
  access libraries do not exist in shared form on such systems.
  There is also likely to be an issue when running with thread-safe
  libraries.

1.48   mpatrol.guide/Amiga notes

```
Notes for Amiga platforms
=========================
```

* Perhaps add support for building mpatrol as an Amiga shared
  library. I attempted to do this in a previous release of mpatrol,
  but it would have involved too many source changes to get working
  fully. Perhaps it's not even worth implementing as the archive
  library works fine.

* Need to add proper support for call stack traversal for both the
  Motorola 680x0 and PowerPC processor architectures. When gcc is
  being used then up to two stack frames can be traversed, but this
  should really be extended without requiring
  MP_BUILTINSTACK_SUPPORT. When SAS/C is being used then there is
  no support for call stack traversal.

* Need to add proper support for reading symbols from Amiga
  executable files. When gcc is being used then the BFD library
  routines will be called to determine the symbols from the
  executable file, but this will only work for objects compiled with
  gcc. When SAS/C is being used then there is no support for
  reading symbols from executable files. Also need to add support
  for reading symbols from any shared libraries that are required by
  the program.

* Possibly make use of other software such as Enforcer, Mungwall or
  MuLib in order to provide some form of memory protection. The
  features of SegTracker could also be put to good use so that the
  file and hunk location of entries on the call stack could be
  determined.

* When using SAS/C it is currently not possible to override the
  definition of malloc(), etc., without including the mpatrol.h
  header file first. This is because the compiler startup code and
  libraries call malloc() before everything is set up, and so the
  library cannot properly initialise itself if the malloc() that the
  startup code finds is the malloc() in the mpatrol library. This
  restriction does not exist when using gcc.

1.49   mpatrol.guide/Windows notes

```
Notes for Windows platforms
===========================
```

   * Need to add watch point area support, possibly by using guard
     pages as a basis for an implementation.

   * Need to add support for reading symbols from Windows executable
     files. Also need to add support for reading symbols from any DLLs
     that are required by the program. This may be possible in a
     limited fashion by using the GNU BFD library, but may only work
     with code compiled with gcc.

   * There seems to be a problem when mixing the archive version of the
     mpatrol library and the Microsoft C run-time library DLL, and vice
     versa. This needs to be looked into, but for the moment, don't
     mix them.

1.50   mpatrol.guide/Netware notes

```
Notes for Netware platforms
===========================
```

   * The library has not yet been built (let alone tested) on Netware
     platforms. The names of the system functions that the library
     calls for Netware were obtained by looking at Novell's developer
     documentation, so they may not even compile correctly without
     modification.

   * Need to add support for building the mpatrol library as an NLM.
     This is not currently a high priority requirement as the archive
     library should suffice for most purposes.

   * Need to add way to determine when the base of the stack has been
     reached during call stack traversal, since on Netware every
     application is really a thread running under one large process.

   * Need to add support for reading symbols from Netware load modules.
     Also need to add support for reading symbols from any NLMs that
     are required by the program. This may be possible in a limited
     fashion by using the GNU BFD library, but may only work with code
     compiled with gcc.

   * Need to investigate if it is safe (or even possible) to override
     the definitions of malloc(), etc., without including the mpatrol.h
     header file first. Currently, non-macro definitions for these
     functions have been disabled in the Netware version of the library
     in case they affect other NLMs that are currently running.

1.51   mpatrol.guide/Related software

Related software
****

   A list of software which helps in debugging dynamic memory
allocation problems is given below(1). They all provide some of the
features that mpatrol contains and you may wish to use one of them to
solve your problem if you have trouble using mpatrol. I have only ever
used Dbmalloc and Electric Fence, so I can't vouch for any of the
others, although if you have any recommendations feel free to let me
know so I can add them to this list. In particular, there seems to be
a shortage of such programs for Netware platforms.

   * APurify
     Author
           Samuel Devulder (<Samuel.Devulder@info.unicaen.fr>)

     License
           Free Software

     Platforms
           AmigaOS

     Location
           'http://wuarchive.wustl.edu/~aminet/dirs/dev_debug.html'

     Overview
           Instruments an assembler source file to insert code that
           checks all memory accesses.

   * BoundsChecker
     Author
           NuMega Corporation (<info@numega.com>)

     License
           Commercial Software

     Platforms
           MS-DOS, Windows

     Location
           'http://www.numega.com/'

     Overview
           Detects and diagnoses errors in static, stack and heap memory
           and in memory and resource leaks.

   * Ccmalloc
     Author
           Armin Biere (<armin@ira.uk.de>)

     License
           GNU General Public License

     Platforms
           Various UNIX

Location
       'http://iseran.ira.uka.de/~armin/ccmalloc/'

Overview
       Can interface with gdb to find memory leaks, multiple
       deallocations and memory corruptions in C or C++ programs.

* Chaperon
 Author
       John Reiser (<jreiser@BitWagon.com>)

 License
       Commercial Software

 Platforms
       Linux

 Location
       'http://www.BitWagon.com/chaperon.html'

 Overview
       Runs existing Intel Linux binary application programs, but
       checks for and reports bad behaviour in accessing memory.

* Checker
 Author
       Tristan Gingold (<bug-checker@gnu.org>)

 License
       GNU General Public License

 Platforms
       Various UNIX

 Location
       'http://www.gnu.org/'

 Overview
       Detects illegal memory accesses when reading from
       uninitialised memory, writing to freed memory or outside
       memory blocks. Also contains a garbage collector for
       detecting memory leaks.

* CSRI malloc
 Author
       Mark Moraes (<moraes@deshaw.com>)

 License
       Free Software

 Platforms
       Various UNIX

 Location
       'ftp://ftp.cs.toronto.edu/pub/moraes/malloc.tar.gz'

 Overview

A library of dynamic memory allocation functions with limited
debugging and profiling support and detection of memory
leaks. Also comes with a graphical tool to display a dynamic
picture of the heap.

* Dbmalloc
 Author
        Conor P. Cahill (<cpcahil@virtech.vti.com>)

 License
        Free Software

 Platforms
        Various UNIX

 Location
        'http://www.clark.net/pub/dickey/dbmalloc/dbmalloc.html'

 Overview
        Provides replacements for memory management library functions
        and provides a full set of debugging features which detect
        memory overruns and other types of misuse.

* Debauch
 Author
        Jon A. Christopher (<jac8792@tamu.edu>)

 License
        GNU General Public License

 Platforms
        Linux

 Location
        'http://quorum.tamu.edu/jon/gnu/'

 Overview
        A memory allocation debugger for C which will detect memory
        leaks, corrupted memory, stores to freed memory and more.

* Debug Heap
 Author
        IBM Corporation (<info@ibm.com>)

 License
        Commercial Software

 Platforms
        IBM AS/400

 Location
        'http://www.as400.ibm.com/developer/porting/heapexternal.html'

 Overview
        A heap debugging environment with stack traceback for IBM
        AS/400 servers.

* Dmalloc
 Author
        Gray Watson (<gray@burger.letters.com>)

 License
        Free Software

 Platforms
        Various UNIX, MS-DOS, Windows

 Location
        'http://www.dmalloc.com/'

 Overview
        A drop-in replacement for the system's memory management
        routines, providing powerful debugging facilities
        configurable at run-time. Formerly known as Malloc_Dbg.

* Electric Fence
 Author
        Bruce Perens (<Bruce@Pixar.com>)

 License
        GNU General Public License

 Platforms
        Various UNIX

 Location
        'ftp://ftp.perens.com/pub/ElectricFence/'

 Overview
        Uses virtual memory hardware to protect dynamically allocated
        memory in order to detect illegal memory accesses.

* Enforcer
 Author
        Michael Sinz (<Enforcer@sinz.org>)

 License
        Free Software

 Platforms
        AmigaOS

 Location
        'http://www.iam.com/amiga/enforcer.html'

 Overview
        Sets up MMU tables to watch for illegal accesses to memory,
        such as the low page and non-existent pages.

* FDA (Free Debug Allocator)
 Author
        Thomas Helvey (<tomh@inxpress.net>)

 License

```
            GNU General Public License

  Platforms
        Linux, Windows

  Location
        'http://www.debian.org/Packages/unstable/devel/fda.html'

  Overview
        Provides routines that can be plugged in to replace malloc(),
        calloc(), realloc() and free().

* Fortify
  Author
        Simon Bullen (<sbullen@cybergraphic.com.au>)

  License
        Free Software

  Platforms
        AmigaOS

  Location




        'http://www.geocities.com/SiliconValley/Horizon/8596/fortify.html'

  Overview
        Provides a fortified shell for memory allocations, trapping
        memory leaks, writes beyond and before memory blocks and
        writes to freed memory.

* GC (Garbage Collector)
  Author
        Hans-J. Boehm (<boehm@acm.org>)

  License
        Free Software

  Platforms
        Various UNIX, AmigaOS, MS-DOS, Windows, MacOS

  Location
        'http://www.hpl.hp.com/personal/Hans_Boehm/gc/'

  Overview
        A general-purpose, garbage-collecting storage allocator that
        is intended to be used as a plug-in replacement for malloc(),
        but can also be used to detect memory leaks.

* GlowCode
  Author
        Electric Software, Inc. (<info@glowcode.com>)
```

```
License
      Commercial Software

Platforms
      Windows

Location
      'http://www.glowcode.com/'

Overview
      Provides a profiler, call coverage tool and resource browser
      which can detail memory leaks.

* Great Circle
Author
      Geodesic Systems (<info@geodesic.com>)

License
      Commercial Software

Platforms
      Various UNIX, Windows

Location
      'http://www.geodesic.com/'

Overview
      Provides complete heap profiling, allowing programmers to see
      what parts of a program are using the most memory with
      symbolic stack tracing.

* HeapAgent
Author
      MicroQuill (<info@microquill.com>)

License
      Commercial Software

Platforms
      Windows

Location
      'http://www.microquill.com/'

Overview
      Instruments the heap to provide heap error detection without
      the need to recompile any source code.

* Insure++
Author
      ParaSoft (<info@parasoft.com>)

License
      Commercial Software

Platforms
```

```
         Various UNIX, Windows

  Location
        'http://www.parasoft.com/'

  Overview
        Uses Source Code Instrumentation and Runtime Pointer Tracking
        technologies to pinpoint memory corruption, memory leaks,
        operations on unrelated pointers and more.  The Inuse
        graphical memory usage display tool is also provided with this
        software.

* JMalloc
 Author
        Jeff Dunlop

  License
        Free Software

  Platforms
        MS-DOS, Windows

  Location
        'http://www.snippets.org/'

  Overview
        Provides tracing and debugging for malloc() and operator new.

* JProbe
 Author
        KL Group (<info@klgroup.com>)

  License
        Commercial Software

  Platforms
        Various UNIX, Windows

  Location
        'http://www.klgroup.com/'

  Overview
        Helps pinpoint memory leaks in Java applications by tracking
        which objects hold references to other objects, and allows
        visualisation of memory usage in real-time.

* Leak
 Author
        Christopher Phillips (<pefv700@hermes.chpc.utexas.edu>)

  License
        Free Software

  Platforms
        Various UNIX

  Location
```

'http://sources.isc.org/devel/memleak/leak.txt'

Overview
        Logs all calls to malloc() and related functions to database
        files with the filename and line number, then attempts to
        validate reallocations and deallocations and detect memory
        leaks.

* LeakTracer
 Author
        Erwin Andreasen (<erwin@andreasen.org>)

 License
        Free Software

 Platforms
        Various UNIX

 Location
        'http://www.andreasen.org/LeakTracer/'

 Overview
        Detects memory leaks in C++ programs by overriding operator
        new and operator delete.

* Leaky
 Author
        Kipp Hickman (<kipp@netscape.com>)

 License
        Netscape Public License

 Platforms
        Linux

 Location
        'http://www.mozilla.org/unix/leaky.html'

 Overview
        A program which helps find memory leaks and helps debug
        reference count problems with xpcom objects.

* Malloc Debug
 Author
        Brandon S. Allbery <allbery@ncoast.org>

 License
        Free Software

 Platforms
        Various UNIX

 Location

'http://www.leo.org/pub/comp/usenet/comp.sources.misc/malloc-debug/'

Overview
       A debugging malloc package with stack traceback capability.

* Malloc Debug Library
 Author
       Rammi (<rammi@quincunx.escape.de>)

 License
       Free Software

 Platforms
       Various UNIX

 Location
       'http://www.escape.de/users/quincunx/rmdebug.html'

 Overview
       Implements wrappers for the normal heap handling functions.

* MallocTrace
 Author
       Mark Brader (<msb@sq.sq.com>)

 License
       Free Software

 Platforms
       Various UNIX

 Location

       'ftp://ftp.uu.net/usenet/comp.sources.unix/volume18/malloc-trace.Z'

Overview
       A malloc package with call stack tracebacks.

* MCheck
 Author
       Ronald Veldema (<rveldema@cs.vu.nl>)

 License
       GNU General Public License

Platforms
        Linux

Location
        'http://www.cs.vu.nl/~rveldema/mcheck/mcheck.html'

Overview
        A memory usage and malloc checker for C and C++. Comes with
        a Java application for browsing the trace files produced.

* MEM
 Author
        Walter Bright

 License
        Free Software

 Platforms
        MS-DOS

 Location
        'http://www.snippets.org/'

 Overview
        A set of functions for debugging pointer and memory
        allocation problems.

* MemCheck
 Author
        Stratosware Corporation (<info@stratosware.com>)

 License
        Commercial Software

 Platforms
        Windows

 Location
        'http://www.stratosware.com/'

 Overview
        Detects various run-time errors related to operating system
        resources and provides information on memory leaks.

* MemDebug
 Author
        Rene Schmit (<rene.schmit@crpht.lu>)

 License
        Free Software

 Platforms
        Various UNIX, MS-DOS, Windows, MacOS

 Location
        'ftp://ftp.crpht.lu/pub/sources/memdebug/'

    Overview
        Provides memory management error detection, memory usage
        error detection, memory usage profiling and error simulation.

* MemLeak
 Author
        Keith Packard (<keithp@ncd.com>)

 License
        Free Software

 Platforms
        Various UNIX

 Location
        `ftp://ftp.x.org/pub/R6.4/xc/util/memleak/'

 Overview
        Replaces the C library allocation functions and provides
        extensive memory checking, locating lost memory, detecting
        free memory still in use and stores to free memory along with
        stack tracebacks.

* Memory Advisor
 Author
        PLATINUM Technology (<info@platinum.com>)

 License
        Commercial Software

 Platforms
        Various UNIX

 Location
        `http://www.platinum.com/'

 Overview
        Disassembles an object module into system-independent
        assembler code, inserts error checking instructions, then
        re-assembles the code. Can also replace existing malloc
        libraries in order to provide greater error checking.
        Formerly known as Sentinel.

* Memory Sleuth
 Author
        TurboPower (<info@turbopower.com>)

 License
        Commercial Software

 Platforms
        Windows

 Location
        `http://www.turbopower.com/'

Overview
      Quickly tracks down memory leaks and resource allocation
      errors with C++Builder and Delphi.

* Memprof
 Author
      Owen Taylor (<otaylor@redhat.com>)

 License
      GNU General Public License

 Platforms
      Linux

 Location
      'http://people.redhat.com/otaylor/memprof/'

 Overview
      A tool for profiling memory usage and detecting memory leaks.

* Memproof
 Author
      AutomatedQA (<info@totalqa.com>)

 License
      Free Software

 Platforms
      Windows

 Location
      'http://www.totalqa.com/'

 Overview
      A memory and resource leak debugger for Borland's family of
      Windows compilers.

* MemWatch
 Author
      Johan Lindh (<johan@link-data.com>)

 License
      Free Software

 Platforms
      Various UNIX, Windows

 Location
      'http://www.link-data.com/'

 Overview
      A fault-tolerant memory leak and corruption detection tool.

* MemWatch
 Author
      Doug Walker (<walker@unx.sas.com>)

```
    License
          Free Software


    Platforms
          AmigaOS


    Location
          'http://wuarchive.wustl.edu/~aminet/dirs/dev_debug.html'


    Overview
          Provides replacement memory allocation routines for adding
          lots of memory debugging features that you link into your
          program.

* MM (Shared Memory Library)
  Author
          Ralf S. Engelschall (<rse@engelschall.com>)


    License
          Free Software


    Platforms
          Various UNIX, Windows


    Location
          'http://www.engelschall.com/sw/mm/'


    Overview
          Simplifies the usage (and can help debug) the use of shared
          memory between related processes.

* Mmalloc
  Author
          Mike Haertel (<mike@ai.mit.edu>) and Fred Fish
          (<fnf@cygnus.com>)


    License
          GNU General Public License


    Platforms
          Various UNIX


    Location
          'http://www.gnu.org/'


    Overview
          Uses mmap() to allocate separate pools of memory which can be
          mapped onto files for later reuse.

* MPR
  Author
          Taj Khattra (<taj.khattra@pobox.com>)


    License
          Free Software


    Platforms
```

```
        Linux

  Location
        'http://metalab.unc.edu/pub/Linux/devel/lang/c/mpr-2.0.tar.gz'

  Overview
        Attempts to find memory leaks in C/C++ programs by writing a
        log file during program execution, which can then be
        processed for obtaining further information.

* Mprof
 Author
        Ben Zorn (<zorn@cs.colorado.edu>)

  License
        Free Software

  Platforms
        Various UNIX

  Location
        'ftp://gatekeeper.dec.com/pub/misc/mprof-3.0.tar.Z'

  Overview
        Profiles the dynamic memory allocation behaviour of programs
        by logging details for each function than makes a memory
        allocation, including call stack tracebacks.

* MuForce
 Author
        Thomas Richter (<thor@einstein.math.tu-berlin.de>)

  License
        Free Software

  Platforms
        AmigaOS

  Location
        'http://www.math.tu-berlin.de/~thor/thor/index.html'

  Overview
        Uses the MMU to monitor the system for any writes to
        non-existent memory and reports them over the serial port or
        any other output stream.

* MuGuardianAngel
 Author
        Thomas Richter (<thor@einstein.math.tu-berlin.de>)

  License
        Free Software

  Platforms
        AmigaOS

  Location
```

            'http://www.math.tu-berlin.de/~thor/thor/index.html'

  Overview
        An extension to the MuForce program which protects free
        memory and detects all illegal memory accesses.

* MuLib
 Author
        Thomas Richter (<thor@einstein.math.tu-berlin.de>)

 License
        Free Software

 Platforms
        AmigaOS

 Location
        'http://www.math.tu-berlin.de/~thor/thor/index.html'

 Overview
        Provides access to the MMU in modern Amigas so that features
        such as virtual memory can be implemented.

* Mungwall
 Author
        Commodore-Amiga, Inc. (<info@amiga.de>)

 License
        Free Software

 Platforms
        AmigaOS

 Location
        'http://wuarchive.wustl.edu/~aminet/dirs/dev_debug.html'

 Overview
        Patches the system to check for free memory corruption.

* NJAMD (Not Just Another Malloc Debugger)
 Author
        Mike Perry (<mikepery@fscked.org>)

 License
        GNU General Public License

 Platforms
        Various UNIX

 Location
        'http://fscked.org/proj/njamd.shtml/'

 Overview
        Helps track down a wide range of memory allocation problems
        and is divided into a front end executable and a library back
        end.

* Optimizeit
 Author
        Intuitive Systems, Inc. (<info@optimizeit.com>)

 License
        Commercial Software

 Platforms
        Various UNIX, Windows

 Location
        'http://www.optimizeit.com/'

 Overview
        Attempts to locate memory leaks and performance bottlenecks
        in Java programs.

* Plumber
 Author
        Owen O'Malley (<omalley@ics.uci.edu>)

 License
        GNU General Public License

 Platforms
        Linux, Solaris, SunOS

 Location
        'http://www.ics.uci.edu/~softtest/plumber.html'

 Overview
        A tool that replaces the normal Ada and C/C++ dynamic memory
        allocation functions and detects unfreed memory blocks.

* Purify
 Author
        Rational Software (<info@rational.com>)

 License
        Commercial Software

 Platforms
        Various UNIX, Windows

 Location
        'http://www.rational.com/'

 Overview
        Uses Object Code Insertion technology to provide run-time
        error checking and memory leak detection.

* QC
 Author
        Onyx Technology (<sales@onyx-tech.com>)

 License
        Commercial Software

    Platforms
          MacOS

    Location
          'http://www.onyx-tech.com/'

    Overview
          Runs in the background as a control panel and detects various
          memory errors which can then be caught and run under a
          debugger.

* TestCenter
  Author
          CenterLine Development Systems (<info@centerline.com>)

  License
          Commercial Software

  Platforms
          Various UNIX

  Location
          'http://www.centerline.com/'

  Overview
          Detects memory leaks, duplicate frees and illegal access
          errors including loads from uninitialised objects.

* Third Degree
  Author
          Digital Equipment Corporation (<info@digital.com>)

  License
          Commercial Software

  Platforms
          Digital UNIX

  Location
          'http://www.digital.com/'

  Overview
          A tool that performs memory access checks and memory leak
          detection of C, C++ and Fortran programs at run-time.
          Applications are modified using ATOM to determine if any
          memory locations are accessed when not properly allocated or
          initialised.

* Vmalloc
  Author
          Kiem-Phong Vo (<kpv@research.att.com>)

  License
          AT&T Source Code License

  Platforms

```
        Various UNIX, Windows

  Location
        'http://akpublic.research.att.com/sw/tools/vmalloc/'

  Overview
        A discipline and method library for dynamic memory
        allocation, with support for regions, debugging and profiling.

* Wipeout
 Author
        Olaf Barthel (<olsen@sourcery.han.de>)

 License
        Free Software

 Platforms
        AmigaOS

 Location
        'http://wuarchive.wustl.edu/~aminet/dirs/dev_debug.html'

 Overview
        Runs in the background checking free memory for corruption.

* YAMD (Yet Another Malloc Debugger)
 Author
        Nate Eldredge (<neldredge@hmc.edu>)

 License
        GNU General Public License

 Platforms
        Linux, DOS

 Location
        'http://www3.hmc.edu/~neldredge/yamd/'

 Overview
        A tool for finding bugs related to dynamic memory allocation
        in C and C++, and includes paging mechanisms to catch bugs
        immediately.

* ZeroFault
 Author
        The Kernel Group (<info@zerofault.com>)

 License
        Commercial Software

 Platforms
        AIX UNIX

 Location
        'http://www.zerofault.com/'

 Overview
```

Uses run-time emulator technology to provide run-time error
checking and memory leak detection.

However, before you try out any of the above software, there may
already be a malloc library with debugging support on your system that
might be suitable for solving your problem. For example, on Solaris 7
the following libraries are available:

'malloc(3c)'
     Trade-off between performance and efficiency.

'malloc(3x)'
     Slower performance, space-efficient.

'bsdmalloc(3x)'
     Better performance, space-inefficient.

'mtmalloc(3t)'
     Thread-safe memory allocator.

'mapmalloc(3x)'
     Uses mmap() instead of sbrk() to allocate heap space.

'watchmalloc(3x)'
     Uses watch point areas to check for overflows.

On platforms with the GNU C library, such as Linux, there are several
environment variables that can be used to enable various debugging
features of malloc(), etc. There are also extra functions provided in
the library which can be used to aid in debugging, and some shell
scripts which can translate return addresses or locate unfreed memory
allocations in the log files produced. Useful information on the
debugging features available within the GNU C library is located at
'http://sdb.suse.de/sdb/en/html/aj_debug.html'.

---------- Footnotes ----------

(1) This list can be considered to be a slightly more up to date
version of 'Debugging Tools for Dynamic Storage Allocation and Memory
Management' ('http://www.cs.colorado.edu/~zorn/MallocDebug.html') by Ben
Zorn (<zorn@cs.colorado.edu>).

1.52 mpatrol.guide/Function index

Function index
****

__mp_check
     Functions

__mp_epilogue
     Functions

1.53   mpatrol.guide/Index

                Index
*****

-1

  Options

-2

  Options

-3

  Options

-A

  Options

-a

  Options

-C

  Options

-c

  Options

-d

  Options

-D

  Options

-e

  Options

-F

  Options

-f

  Options

-g

  Options

-G

  Options

-L

  Options

-l

  Options

-M

  Options

-m

  Options

-N