

version 1.0.0
January 2005

Elephant

A C++ Memory Observer

User Guide



Paul Grenyer
paul (at) paulgrenyer.co.uk

<http://www.paulgrenyer.dyndns.org/elephant/>

Table of Contents

Elephant.....	1
What is Elephant?.....	3
Where can I get Elephant?.....	3
What do I need to build Elephant?.....	3
How do I build Elephant?.....	3
Microsoft Visual C++ 7.1.....	4
MinGW.....	4
g++.....	4
Xcode 1.5.....	5
How do I setup my environment to use Elephant?.....	5
Microsoft Visual C++ 7.1.....	6
MinGW & g++.....	7
Xcode 1.5.....	7
How do I use Elephant in my program?.....	8
operator new and operator delete.....	8
Example 1: Observing and reporting a memory leak.....	9
Example 2: Recording line and filename of allocation.....	11
Example 3: Using the maximum memory observer.....	12
Example 4: Writing a custom memory observer (part 1).....	13
Example 5: Writing a custom memory observer (part 2).....	16
Elephant and Threading.....	19
Elephant Mutexes.....	19
Custom Mutexes.....	20
Where Next.....	21

What is Elephant?

Elephant is a C++ memory observer. It keeps track of all calls to `new` and `delete` via custom implementations of `operator new` and `operator delete`. Observers can register to be notified of allocations and deletions and used to detect memory leaks, keep a track of maximum memory usage or for any other purpose, by implementing a simple interface.

A notification of an allocation consists of the address and size of the memory allocated. The line number, function name and file name in which the allocation takes place can be added by placing special macros in the client code. A notification of a deletion consists of the address of the memory being freed.

Elephant is not intended to ship in production code. It is intended as a debugging aid. Elephant's functionality can be removed simply by relinking without the Elephant static library. All other code can remain in place.

Elephant comes with a complete, Aeryn (<http://www.paulgrenyer.co.uk/aeryn>) based test suite to test that it behaves correctly on any given platform.

Where can I get Elephant?

Elephant is available for download from:
<http://www.paulgrenyer.dyndns.org/elephant/>

What do I need to build Elephant?

Elephant uses up-to-date C++ techniques (including member function templates using the Aeryn unit tests), as well as some classes based on parts of Andrei Alexandrescu's Loki library (<http://sourceforge.net/projects/loki-lib/>) and therefore requires a modern compiler. It has been tested on, and provides make files or project files for the following compilers:

Microsoft Visual C++ 7.1
MinGW 3.2.3
GNU G++ 3.2.3 & 3.4.3
Xcode 1.5

It *may* be possible to get Elephant to compile on Microsoft Visual C++ 6.0.

How do I build Elephant?

Elephant consists of a group of headers and static library. The full source is supplied with Elephant and the static library must be built. Building the elephant static library couldn't be easier:

Microsoft Visual C++ 7.1

To build the Elephant library, unit tests and the (test) supporting Aeryn library with Microsoft Visual C++ 7.1, simply open the Elephant solution located in the top level Elephant directory and select Build Solution from the Build menu.

To run the unit tests right click on the TestClient project in the Solution Explorer and select Set as StartUp Project, then select Start Without Debugging from the Debug menu. This should give you the following output:

```
Aeryn 0.4.0 beta (c) Paul Grenyer 2004
http://www.paulgrenyer.co.uk/aeryn
-----

Ran 21 tests, 21 Passed, 0 Failed.

Press any key to continue
```

MinGW

To build the Elephant library, unit tests and the (test) supporting Aeryn library with MinGW open a command prompt and navigate to the top level Elephant directory. Making sure that the MinGW bin directory is in your path, type:

```
mingw32-make
```

To run the unit tests type the following:

```
bin\TestClient.exe
```

This should give you the following output:

```
Aeryn 0.4.0 beta (c) Paul Grenyer 2004
http://www.paulgrenyer.co.uk/aeryn
-----

Ran 21 tests, 21 Passed, 0 Failed.
```

For mingw32-make clean to work correctly the rm tool from MSYS or cygwin must also be in your path.

g++

To build the Elephant library, unit tests and the (test) supporting Aeryn library with g++ open a command prompt and navigate to the top level Elephant directory. Checking that g++ and make are both installed correctly, type:

```
make
```

To run the unit tests type the following:

```
bin/TestClient.exe
```

This should give you the following output:

```
Aeryn 0.4.0 beta (c) Paul Grenyer 2004
http://www.paulgrenyer.co.uk/aeryn
-----
Ran 21 tests, 21 Passed, 0 Failed.
```

The current version of Elephant was tested with g++ 3.2.3 on Red Hat Linux ES 3.0. If any of the tests fail on your platform Elephant may not work as expected. If you do have tests that fail, please send me the complete Aeryn output along with details of your g++ version and operating system.

Xcode 1.5

To build the Elephant library, unit tests and the (test) supporting Aeryn library with Xcode 1.5, simply open the Elephant Xcode project located in the top level Elephant directory and Build the ElephantLib target via the Build menu.

The unit tests are built and run automatically as part of the build process. Any failed tests will appear as build errors.

Note: Building Elephant manually like this for Xcode is not required. ElephantLib will be build automatically when following the instructions in the section: "How do I setup my environment to use Elephant?".

How do I setup my environment to use Elephant?

Before you can use Elephant the Elephant static library must be built (see previous section):

Compiler	Library name
Microsoft Visual C++ 7.1	Elephant_debug.lib (debug) Elephant.lib (release)
MinGW	libelephant.a
g++	libelephant.a
Xcode 1.5	libelephant.a

Regardless of which compiler or platform is used the Elephant library is places in the bin directory which a subdirectory of the Elephant top level directory.

Your environment also needs to have access to the Elephant include directory which is a subdirectory of the top level Elephant directory. The actual Elephant include files are stored in further subdirectories called elephant and tools (tools is a subdirectory of elephant). This is so that Elephant include files can be identified from other include files which might share the same name. For example:

```
#include <elephant/newdelete.h>
```

Microsoft Visual C++ 7.1

Once you have created a solution containing the project which is going to use Elephant to monitor memory usage, you are ready to add Elephant to your environment.

There are at least two ways to add the Elephant static library to your solution:

Method 1: Add the ElephantLib project to the solution.

This method has the advantage that the ElephantLib project is included in a rebuild all.

1. Right click the solution name in Solution Explorer and select Add Existing Project from the Add menu item.
2. Navigate to the ElephantLib directory which is a subdirectory of the Elephant top level directory.
3. Select ElephantLib.vcproj and click open. (This will add the Elephant library project to your solution.)
4. Right click your project and select Project Dependencies from the menu. Then put a tick in the ElephantLib box and click Ok.

Method 2: Add the Elephant static library directly to the project.

1. Right click your project and select properties.
2. Set the Configuration drop-down box to All Configurations.
3. Select the Linker folder and then the General item in the tree view.
4. Enter the path to the Elephant static libraries (Elephant\bin) into the Additional Library Directories box.
5. Set the Configuration drop-down box to debug.
6. Select the Input item from the Linker folder in the tree view.

7. Enter Elephant_debug.lib into the Additional Dependencies box.
8. Set the Configuration drop-down box to release.
9. Enter Elephant.lib into the Additional Dependencies box.
10. Click Ok

To make the Elephant headers available to your project in your solution follow these steps:

1. Right click your project and select properties.
2. Set the Configuration drop-down box to All Configurations.
3. Select the C/C++ folder and then the General item in the tree view.
4. Enter the path to the Elephant include files (Elephant\include) into the Additional Include Directories box.
5. Click Ok.

MinGW & g++

This description of configuring MinGW and g++ to link to Elephant assumes that you are using a make file to build your project. Of course this is not the only way.

To link Elephant to your executable (or shared library etc) two extra parameters need to be added to your link command: the path to the Elephant static library, preceded by `-L` and the name of library, preceded by `-l`. For example:

```
g++ myproj.o -LElephant/bin -lelephant myproj
```

The Elephant include files must be made available to every invocation of g++ that builds a source (cpp) file that includes, directly or indirectly, an Elephant include file. This is done by adding a single parameter, which consists of the path to the Elephant include directory preceded by `-I`. For example:

```
g++ -c -o myproj.o myproj.cpp -IElephant/include
```

Xcode 1.5

Project Changes

- Add elephant.xcode to project
(Add -> Existing Files)
- Turn of ZeroLink
(Get Info -> Styles -> Linking -> ZeroLink)

Target Changes

For any target in the project that uses Elephant:

- Drag `libelephant.a` into the Target's Frameworks & Libraries
- Add a dependency on the Elephant project `ElephantLib` target
(Get Info -> General -> [+])
- Add path to (`elephant/include`) to header search paths
(Get Info -> Build -> Search Paths -> Header Search Paths)
- Add path to (`elephant/bin`) to library search paths
(Get Info -> Build -> Search Paths -> Library Search Paths)

How do I use Elephant in my program?

Assuming that you have built the Elephant static library and integrated it into your environment (see previous two sections) you are now ready to use Elephant in your program.

`operator new` and `operator delete`

The custom implementations of `operator new` and `operator delete` are the key to Elephant's ability to monitor memory. There are overloads for the normal and array versions with corresponding no throw versions.

To use the Elephant's custom new and delete operators simply include the `newdelete.h` header in your program. For example:

```
#include <elephant/newdelete.h>

int main()
{
    return 0;
}
```

It only needs to be included once, although multiple inclusions will not do any harm.

Every time a call is made to new or delete the Elephant operator overloads will register the call with the Elephant memory monitor. The Elephant memory monitor is observable and you can register one of the provided observers or write your own to react to the allocations and de-allocations.

Example 1: Observing and reporting a memory leak.

Let's start with a simple example of a memory leak:

```
#include <elephant/newdelete.h>

class SomethingToAllcoate
{};

int main()
{
    SomethingToAllcoate* p = new SomethingToAllcoate;
    return 0;
}
```

This program will compile and run and you will see absolutely no indication of the memory leak. In order to detect the memory leak you need the leak detector class, `LeakDetector`. The leak detector class is an observer of the memory monitor, so you need to register and unregister it as an observer:

```
#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/leakdetector.h>

class SomethingToAllcoate
{};

int main()
{
    using namespace elephant;
    LeakDetector leakDetector;

    // Register leak detector with memory monitor.
    MemoryMonitorHolder().Instance().
        AddObserver( &leakDetector );

    SomethingToAllcoate* p = new SomethingToAllcoate;

    // Unregister leak detector with
    // memory monitor.
    MemoryMonitorHolder().Instance().
        RemoveObserver( &leakDetector );

    return 0;
}
```

To use the memory monitor and the leak detector you need to include the appropriate header files as shown. Running this program will still not indicate that there is a memory leak. To indicate the memory leak you need to interrogate the `LeakDetector` instance. For example:

```

#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/leakdetector.h>
#include <cassert>

class SomethingToAllcoate
{};

int main()
{
    using namespace elephant;
    LeakDetector leakDetector;

    // Register leak detector with memory monitor.
    MemoryMonitorHolder().Instance().
        AddObserver( &leakDetector );

    SomethingToAllcoate* p = new SomethingToAllcoate;

    // Unregister leak detector with
    // memory monitor.
    MemoryMonitorHolder().Instance().
        RemoveObserver( &leakDetector );

    assert( !leakDetector.IsLeak() );
    return 0;
}

```

The `assert` (which required the `cassert` header as shown) will indicate that a memory leak has occurred. This particular method of indicating a memory leak isn't particularly useful. The next step is to print the memory address and the size of the leak:

```

#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/leakdetector.h>
#include <elephant/leakdisplayfunc.h>
#include <algorithm>

class SomethingToAllcoate
{};

int main()
{
    using namespace elephant;
    LeakDetector leakDetector;

    // Register leak detector with memory monitor.
    MemoryMonitorHolder().Instance().
        AddObserver( &leakDetector );

    SomethingToAllcoate* p = new SomethingToAllcoate;

    // Unregister leak detector with
    // memory monitor.
    MemoryMonitorHolder().Instance().
        RemoveObserver( &leakDetector );
}

```

```

        // Display the details of the leak.
        LeakDisplayFunc leakDisplay( std::cout );
        std::for_each(    leakDetector.begin(),
                        leakDetector.end(),
                        leakDisplay );

        return 0;
    }

```

The `LeakDisplayFunc` class constructor takes a reference to an output stream and has a function operator that can be used, as shown, to write memory leak information to the stream. As `LeakDisplayFunc` uses an output stream it is possible that memory will be allocated and not freed until the end of `main`. This is why the leak detector must be unregistered before the memory leak information is displayed. Otherwise the output stream allocation will appear as a further memory leak. One way to avoid having to unregistered the `LeakDetector` is to write your own function object that displays the memory leak information without allocating memory using `new`. For example using `printf`.

The output from this program should be as follows, although the address will be a different value:

```

        Address:          00320B70
        Size:             1

```

Example 2: Recording line and filename of allocation.

In the previous example the memory leak was displayed as a memory address and a size. This can be useful in finding a memory leak, but not as usual as tracking the exact site of the allocation. Elephant can do this by introducing a special macro into every translation unit where this type of tracking is needed. The macro is called `ELEPHANTNEW` and can be included anywhere in the translation unit. The following code shows how the macro would be added to example 1:

```

#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/leakdetector.h>
#include <elephant/leakdisplayfunc.h>
#include <algorithm>

#define new ELEPHANTNEW

class SomethingToAllcoate
{
};

int main()
{
    ...
}

```

The output should now look something like this:

```
Address:      00322878
Size:         1
Line:         22
Function:     main
File:         c:\...\example2\main.cpp
```

Some compilers, such as Microsoft Visual C++ 7.1 will show a fully qualified function name and a complete a full file path. Other compilers, such as g++ and MinGW will show only the local function name and file name without the full path. For example:

```
Address:      0x3d24f0
Size:         1
Line:         23
File:         main.cpp
```

Example 3: Using the maximum memory observer

The other memory observer supplied with Elephant, `MaxMemoryObserver`, is for measuring the maximum amount of memory used at anyone time by an application. Its use is very similar to that of `LeakDetector`:

```

#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/maxmemoryobserver.h>

class SomethingToAllcoate
{};

int main()
{
    using namespace elephant;
    MaxMemoryObserver maxMemory;

    // Register max memory observer with memory monitor.
    MemoryMonitorHolder().Instance().
        AddObserver( &maxMemory );

    SomethingToAllcoate *p1 = new SomethingToAllcoate[100];
    delete[] p1;

    SomethingToAllcoate *p2 = new SomethingToAllcoate[50];
    delete[] p2;

    // Unregister max memory observer with
    // memory monitor.
    MemoryMonitorHolder().Instance().
        RemoveObserver( &maxMemory );

    // Display the max memory usage
    std::cout << "Max memory usage: "
        << static_cast< unsigned long >
            ( maxMemory.MaxMemory() )
        << " bytes\n";

    return 0;
}

```

The output from this simple (not very exception safe) example is as follows:

```
Max memory usage: 100 bytes
```

The size of `SomethingToAllcoate` is 1 byte. During the execution of the program a total of 150 `SomethingToAllcoate` instances are created and destroyed. However, the program only has up to 100 instances allocated at any one time. Therefore the maximum amount of memory used by the program is 100 bytes.

Example 4: Writing a custom memory observer (part 1)

Elephant can be used for more than just detecting memory leaks and the maximum memory used by a program. Elephant can be used to monitor any characteristic of new and delete based memory usage via custom memory observers. Custom memory observers are simple to create. All that is required is the implementation of the following interface:

```

namespace elephant
{
    class IMemoryObserver
    {
    protected:
        IMemoryObserver();

    public:
        virtual ~IMemoryObserver() = 0;

        virtual void OnAllocate
            ( void* p,
              std::size_t size,
              std::size_t line,
              const char* file ) = 0;

        virtual void OnFree
            ( void* p ) = 0;

    private:
        IMemoryObserver( const IMemoryObserver& );
        IMemoryObserver& operator=
            ( const IMemoryObserver& );
    };
}

```

All that needs to be done to implement the interface is to inherit from it and override the `OnAllocate` and `OnFree` pure virtual member functions. The `OnAllocate` function has the following arguments:

- `p` A pointer to the memory that has been allocated. This is useful for getting the address.
- `size` The size of the memory that has been allocated.
- `line` The line number on which the memory was allocated. This is 0 unless the `ELEPHANTNEW` macro has been used correctly.
- `char` The file in which the memory was allocated. This is an empty string unless the `ELEPHANTNEW` macro has been used correctly.

The `OnFree` function has the following argument:

- `p` A pointer to the memory that has been allocated. This is useful for getting the address.

The default constructor of the interface is protected to show that the class should be inherited from. The copy constructor and assignment operator are private to prevent attempts to copy the interface or its subclasses (unless the subclasses define their own copy constructor and assignment operator) and the destructor is virtual to ensure proper destruction should a dynamically allocated subclass be destroyed via a pointer to the interface.

The example below is of a simple custom observer which records the total memory allocated by a program during its lifetime:

```
class TotalMemoryobserver : public elephant::IMemoryObserver
{
private:
    std::size_t totalMemory_;

public:
    TotalMemoryobserver()
        : totalMemory_( 0 )
    {
    }

    virtual void OnAllocate
        ( void* p,
          std::size_t size,
          std::size_t line,
          const char* file )
    {
        totalMemory_ += size;
    }

    virtual void OnFree( void* p )
    {
    }

    std::size_t TotalMemory() const
    {
        return totalMemory_;
    }
};
```

The `OnAllocate` override is used to accumulate the size of every allocation. The other parameters are ignored as they are not needed. The `OnFree` function does nothing as we are not interested in de-allocations. In, for example, the leak detector, the value of `p` passed to `OnFree` is used to match against a previous value of `p` passed to `OnAllocate` to show that the memory has been deleted.

Replacing `MaxMemoryObserver`, from the previous example, with `TotalMemoryobserver` and making a couple of other minor changes:

```

int main()
{
    using namespace elephant;
    TotalMemoryobserver totalMemory;

    // Register max memory observer with memory monitor.
    MemoryMonitorHolder().Instance().
        AddObserver( &totalMemory );

    SomethingToAllcoate *p1 = new SomethingToAllcoate[100];
    delete[] p1;
    SomethingToAllcoate *p2 = new SomethingToAllcoate[50];
    delete[] p2;

    // Unregister max memory observer with
    // memory monitor.
    MemoryMonitorHolder().Instance().
        RemoveObserver( &totalMemory );

    // Display the max memory usage
    std::cout << "Total memory usage: "
              << static_cast< unsigned long >
                ( totalMemory.TotalMemory() )
              << " bytes\n";

    return 0;
}

```

gives the following output, which correctly indicates the total memory used by the program:

```
Total memory usage: 150 bytes
```

Example 5: Writing a custom memory observer (part 2)

Sometimes you want to store information about allocations and de-allocations in a container within a custom memory observer. Containers do of course allocate memory in order to contain. This could lead to erroneous memory usage observations and, in a worst case scenario, infinite recursion.

The simple answer is to use a container that uses `malloc` and `free` instead of `new` and `delete`. Or, to be more precise, a container that uses an allocator that allocates with `malloc` and `free` instead of `new` and `delete`. Elephant comes with just such an allocator, called `malloc_allocator`, which can be used with any of the C++ standard library containers. It should be used as follows:

```

#include <elephant/tools/mallocallocator.h>
#include <vector>

...

std::vector< std::size_t,
            elephant::tools::malloc_allocator< std::size_t > >
            allocStore;

```

Naturally a typedef can make life a lot easier.

The following example shows a custom memory observer that uses a container with the `malloc_allocator` to store two lists of the addresses, allocations and deallocations:

```
#include <elephant/newdelete.h>
#include <elephant/memorymonitorholder.h>
#include <elephant/imemoryobserver.h>
#include <elephant/tools/mallocallocator.h>
#include <vector>

class AllocationMemoryobserver : public
elephant::IMemoryObserver
{
private:
    typedef std::vector< void* ,
        elephant::tools::malloc_allocator< void* > >
        MAllocContainer;

    typedef MAllocContainer::const_iterator const_iterator;

    MAllocContainer allocations_;
    MAllocContainer deallocations_;

    void Print
    ( const MAllocContainer& cont, std::ostream& out )
    {
        const_iterator current = cont.begin();
        const_iterator end = cont.end();
        for( ; current != end; ++current )
        {
            out << "\t" << (*current) << "\n";
        }
        out << "\n";
    }

public:
    AllocationMemoryobserver()
        : allocations_(), deallocations_()
    {
    }

    virtual void OnAllocate
    ( void* p,
      std::size_t size,
      std::size_t line,
      const char* file )
    {
        allocations_.push_back( p );
    }

    virtual void OnFree( void* p )
    {
        deallocations_.push_back( p );
    }
};
```

```

void PrintAllocations( std::ostream& out )
{
    out << "Allocations:\n";
    Print( allocations_, out );
}

void PrintDeallocations( std::ostream& out )
{
    out << "Deallocations:\n";
    Print( deallocations_, out );
}

};

class SomethingToAllcoate
{};

int main()
{
    using namespace elephant;
    AllocationMemoryobserver allocationObserver;

    // Register max memory observer with memory monitor.
    MemoryMonitorHolder().Instance().
        AddObserver( &allocationObserver );

    SomethingToAllcoate *p1 = new SomethingToAllcoate[100];
    SomethingToAllcoate *p2 = new SomethingToAllcoate[50];
    delete[] p2;
    delete[] p1;

    // Unregister max memory observer with
    // memory monitor.
    MemoryMonitorHolder().Instance().
        RemoveObserver( &allocationObserver );

    allocationObserver.PrintAllocations( std::cout );
    allocationObserver.PrintDeallocations( std::cout );

    return 0;
}

```

The output from this example is as follows:

```

Allocations:
    00322850
    00322910

Deallocations:
    00322910
    00322850

```

If `malloc_allocator` is replaced by the default allocator, there is no output, not even an error message, with both Microsoft Visual C++ and MinGW.

Elephant and Threading

Elephant has not yet been tested in a multithreaded environment.

The use of the `Mutex` class and its various implementations are based on previously known working examples.

Offers to test Elephant in a multithreaded environment will be gratefully accepted.

By default, Elephant is not thread safe. The `mutex.h` header file is included in a number of places and the `Mutex` class, along with the `Guard` class (for exception safety) is used to protect those parts of the library that may cause problems if accessed by two threads at the same time.

Elephant Mutexes

If you open the `mutex.h` header file, you will see it looks like this:

```
#ifndef ELEPHANT_TOOLS_MUTEX_H
#define ELEPHANT_TOOLS_MUTEX_H

#include <elephant/tools/nullmutex.h>
//#include <elephant/tools/boostmutex.h>
//#include <elephant/tools/win32mutex.h>

#endif // ELEPHANT_TOOLS_MUTEX_H
```

There are three types of mutex supplied with Elephant:

- | | |
|-------------|--|
| Null Mutex | An empty mutex class intended for use in single threaded programs so that no performance is lost creating, entering or leaving an unnecessary mutex. |
| Win32 Mutex | A mutex implemented using the Win32 API intended for use with Windows compilers only. |
| Boost Mutex | A mutex implemented using <code>boost::mutex</code> (http://boost.org/libs/thread/doc/mutex_concept.html). |

The Null Mutex is used by default. To use one of the other mutexes simply include its header file in `mutex.h` *instead* of `nullmutex.h` and rebuild (a rebuild all is recommended).

Custom Mutexes

A custom mutex can be written simply by implementing the following class in its own header file and including it in `mutex.h` *instead* of the other mutex header files:

```
namespace elephant
{
    namespace tools
    {
        class Mutex
        {
        public:
            Mutex()
            {
            }

            ~Mutex()
            {
            }

            void Enter() const
            {
            }

            void Leave() const
            {
            }

        private:
            Mutex( const Mutex& );
            Mutex& operator=( const Mutex& );
        };
    }
}
```

Note: As the `Enter` and `Leave` member functions are `const`, you may need to make the object that holds the current state of the mutex mutable.

Where Next

This is the very first beta release of Elephant. Therefore I expect I, and hopefully other people, will find plenty of bugs or new features that should be implemented, over the coming months.

So far, planned for future releases:

- Threading testing and unit tests.
- Black and white allocation lists
- Client memory tracking