Technical University of Moldova

# MEMORY MANAGEMENT IN THE D PROGRAMMING LANGUAGE

**Student:**

**Panteleev Vladimir**

**Advisor:**

**Lect. Sup. Melnic Radu**

**Chişinău – 2009**

**Ministerul Educaţiei şi Tineretului al Republicii Moldova**

**Universitatea Tehnică a Moldovei**
**Facultatea de Calculatoare, Informatică şi Microelectronică**
**Catedra Filiera Anglofonă**

**Admis la susţinere**
**Şef de catedră: conf. unif. dr. Bostan Viorel**
_____
**„__”_____ 200_**

# Memory Management in the D Programming Language

## Proiect de licenţă

**Student:** _____ (Panteleev V. )
**Conducător:**_____(Melnic R. ___)
**Consultanţi:** _____ (Bostan V.___)
_____ (Nicolenco V. )
_____ (Guţu Al. ____)

**Chişinău – 2009**

## Abstract

This report describes a study of automatic memory management techniques, their implementation in the D Programming Language, and work to improve the state of memory management in the D Programming Language.

**Chapter 1** describes garbage collection as a form of automatic memory management. Automatic memory management implies freeing the programmer of the task of manually managing system memory. This approach has several advantages, such as less programming effort, as well as some disadvantages.

**Chapter 2** outlines the D programming language. D is based on C/C++, however it breaks backwards compatibility in favor of redesigned features, as well as introducing new features present in other modern languages. One of these features is automatic memory management using garbage collection, which is also described in detail.

**Chapter 3** contains three problems commonly encountered while authoring programs in the D programming language (and possibly other garbage collected languages). The covered problems are inadequate performance, memory corruption and memory leaks. The problems are analyzed, and appropriate solutions are developed and deployed.

**Chapter 4** introduces Diamond, a memory debugger and profiler developed specifically for the D programming language. The chapter includes a description of the debugger's features, its inner structure, and two examples of practical application.

**Chapter 5** describes the economic aspect of the project. Since the developed software project is open-source and is available for a free download from the project's website, it has no immediate commercial value.

**Chapter 6** presents several points regarding labor and environment protection.

**Chapter 7** relates about some further directions the project may follow in the future.

## Rezumat

Acest raport descrie un studiu în metode de gestionare automată a memoriei, impementarea acestora în limbajul de programare D, şi efortul de a îmbunătaţi starea curentă a gestionării memoriei în implementarea curentă a limbajului de programare D.

**Capitolul 1** descrie „colectarea reziduurilor” ca o formă de gestionare automată a memoriei. Gestionarea automată a memoriei implică eliberarea programatorului de necesitatea de a gestiona memoria manual. Această metodă are câteva avantaje, cum ar fi micşorarea efortului depus de programatori, dar şi unele dezavantaje.

**Capitolul 2** descrie pe scurt limbajul de programare D. D este bazat pe C/C++, dar el nu păstrează compatibilitatea cu aceste limbaje, în favoarea revizuirii unor elemente a limbajului, şi de asemenea introducerea unol elemente noi. Unul din elementele noi a limbajului introduse în D este gestionarea automată a memoriei, folosind un „colector de reziduuri”.

**Capitolul 3** conţine trei probleme des întâlnite pe parcursul scrierii aplicaţiilor în limbajul D (şi posibil alte limbaje cu gestionare automată a memoriei). Problemele descrise sunt: performanţă neadecvată, corupţie de memorie şi scurgere de memorie. Problemele sunt analizate, şi sunt implementate şi publicate soluţii pentru aceste probleme.

**Capitolul 4** introduce proiectul „Diamond”, un depanator de memorie creat specific pentru limbajul de programare D. Capitolul include o descriere a funcţiilor depanatorului, structura sa internă, şi două exemple de aplicare în practică.

**Capitolul 5** descrie aspectul economic al proiectului. Din cauza că proiectul este publicat ca „open-source” (cu codul sursă disponibil) şi poate fi descărcat de pe pagina web a proiectului, el nu are valoare comercială imediată.

**Capitolul 6** prezintă câteva teme privind protecţia muncii şi a mediului ambiant.

**Capitolul 7** descrie unele direcţii în care proiectul poate fi dezvoltat în continuare.

# Table of Contents

## Introduction

In today's day and age, computer science is rapidly evolving. Hardware developments are relentlessly following Moore's Law, and software evolves at an even faster pace. Millions of lines of source code are written every year around the world. New languages are being invented, aimed at simplifying the task of software development.

One of the languages that is recently becoming more popular is the D Programming Language. The D programming language is designed as a successor to C++ – it inherits the best design decisions, while bringing in new features that give the language more power and flexibility, while reducing the risk of making mistakes which could lead to hours of debugging. Quoting the language's website [5]:

> *D is a general purpose systems and applications programming language. It is a higher level language than C++, but retains the ability to write high performance code and interface directly with the operating system API's and with hardware. D is well suited to writing medium to large scale million line programs with teams of developers. D is easy to learn, provides many capabilities to aid the programmer, and is well suited to aggressive compiler optimization technology.*

One of the more unique aspects of D is how it handles memory allocation. D is a garbage-collected language, which is unusual for a compiled language. Namely, D uses a modified Hans Boehm garbage collector, which is a non-moving, stop-the-world mark-and-sweep garbage collector.

Garbage collection removes the burden of manual memory management from the programmer, however it is not a panacea. Using memory in a way which is not compatible with the garbage collector design may have adverse effects, such as memory leaks or memory corruption. This new approach to memory management requires special considerations and additional research to be utilized in full.

This paper describes a study of memory management in D, as well as several achievements in improving memory-related aspects of software development in D and related contributions to the D community.

# 1 Garbage Collection

In computer science, the term “garbage collection” refers to a technique of automatic memory management. In most cases it implies that allocated objects do not have to be deallocated manually by the programmer, but the environment will automatically destroy the object when it is no longer referenced. The name describes the process of “collecting” (reclaiming memory used by) “garbage” (objects no longer used by the program).

## 1.1 History

Garbage collection was invented by John McCarthy in 1959 for the Lisp programming language. McCarthy introduced garbage collection (called “memory reclamation” in the official paper) to solve the memory problems of Lisp programs [9].

Languages with explicit memory management, such as C and FORTRAN, remained popular for a long time – until Sun introduced Java, which quickly became the most used virtual-machine-based language. The Java platform, followed by its Microsoft clone .NET, used automatic memory management by means of garbage collection. The quality of a garbage collector’s implementation greatly affects a platform’s performance. Ever since Java was launched in 1991, Sun continued to work on the platform’s performance, and even today effort is being done for implementing a more efficient garbage collector: Sun has recently announced the G1 (Garbage First) garbage collector, in JDK 1.6.0 [6].

As computers became faster, performance became less of an issue, and development effort becomes more important than software performance. This brought popularity to many interpreted languages with automatic memory management, such as PHP, Python and Ruby – although these languages’ performance does not scale anywhere near as, for example, C++ programs, software written in them usually requires less lines of code and less debugging. Nowadays, many companies find that it’s cheaper to buy more server hardware than to hire expert C++ programmers.

In 1992, Hans Boehm published a garbage collector implementation for C/C++ [3]. The collector constituted of a library, which replaced the standard memory allocation functions, like C’s “malloc” and C++’s “new”. Since C/C++ compiled to machine code, it

wasn’t possible to track object references directly, Boehm’s garbage collector used a new approach for detecting references.

The D Programming Language is designed to function on a garbage collector following the basic design of Boehm’s GC. Since automatic memory management is actually part of the language design, it allows introducing several language features which depend on it, such as liberal copying of variables without having to burden the programmer of tracking them. This allows D to take the best of compiled and interpreted languages – retaining high performance while providing flexibility and features which reduce software development time.

### 1.2 Advantages

Automatic memory management in the form of garbage collection has several advantages over manual memory management.

The first and most obvious one is relieving the programmer from the effort of manually tracking object references. In complex C/C++ programs, memory management can become very complicated, and mistakes can lead to several categories of problems:

a) **memory leaks** – a memory leak appears when objects which are no longer used by the program are not being deallocated. If this happens periodically during the runtime of the program, the program will continue to unjustly consume increasing amounts of memory, eventually crashing the program or the entire system [8].

b) **dangling pointer bugs** – a “dangling” pointer is a pointer to an object which has been deallocated. This problem happens when an object has been freed in one part of the code, however a reference to the object remains in another location. When the program attempts to reference that pointer, it will in fact reference an unallocated memory region. Sometimes, this memory region will already be allocated for another object – and writing to this reference will result in memory corruption of that object. Such bugs are very dangerous because they can lead to sporadic memory corruption, which will usually crash the program in a completely unrelated point of execution. These bugs are also generally hard to detect and track down –

programmers need to resort to using specialized debugging tools to be able to find the source of these problems.

c) **double free bugs** – a “double free” bug happens when the program attempts to deallocate a memory region which is already free. If another object happens to have been allocated at that address between the two deallocations, that object will be freed instead. Depending on the memory management code, this may lead to undefined behavior and memory corruption.

Automatic memory management allows programmers to worry less about bugs and concentrate on the task at hand. As such, it shortens software development time, reduces development costs and results in higher-quality software.

### 1.3 Performance

A common misconception is that garbage collection is much slower than explicit memory management. However, this is not true. In fact, garbage-collected programs are actually faster. This may sound counter-intuitive, but the reasons are:

a) With explicit memory management, the memory manager is called for every time an object is freed. This involves doing repeated calculations for every deallocation – such as looking up in which memory page is the object allocated, adding the object to a free list, etc. On contrast, a mark-and-sweep garbage collector can greatly optimize these operations, because it operates on large sets of objects simultaneously.

b) With explicit memory management, the programmer needs to write destructors for class objects to explicitly deallocate any “child” objects created by the object being destroyed. This adds an additional overhead – in garbage-collected programs, destructors are written usually to free up OS resources, and most classes don’t have destructors at all.

c) All those destructors freeing memory can become significant when objects are allocated on the stack. For each one, some mechanism must be established so that if an exception happens, the destructors all get called in each frame to release any

memory they hold. If the destructors become irrelevant, then there's no need to set up special stack frames to handle exceptions, and the code runs faster.

d) All the code necessary to manage memory can add up to quite a bit. The larger a program is, the less in the cache it is, the more paging it does, and the slower it runs.
e) Garbage collection kicks in only when memory gets tight. When memory is not tight, the program runs at full speed and does not spend any time freeing memory.
f) Modern garbage collectors are far more advanced now than the older, slower ones. Generational, copying collectors eliminate much of the inefficiency of early mark and sweep algorithms.
g) Modern garbage collectors do heap compaction. Heap compaction tends to reduce the number of pages actively referenced by a program, which means that memory accesses are more likely to be cache hits and less swapping.
h) Garbage collected programs do not suffer from gradual deterioration due to an accumulation of memory leaks.

### 1.4 Downsides

Garbage collection is not a panacea. There are some downsides:

a) It is not predictable when a collection gets run, so the program can arbitrarily pause.
b) The time it takes for a collection to run is not bounded. While in practice it is very quick, this cannot be guaranteed.
c) All threads other than the collector thread must be halted while the collection is in progress.
d) Garbage collectors can keep around some memory that an explicit deallocator would not. In practice, this is not much of an issue since explicit deallocators usually have memory leaks causing them to eventually use far more memory, and because explicit deallocators do not normally return deallocated memory to the operating system anyway, instead just returning it to its own internal pool.
e) Garbage collection should be implemented as a basic operating system kernel service. But since they are not, garbage collecting programs must carry around with

them the garbage collection implementation. While this can be a shared DLL, it is still there.

### 1.5 Classification

Garbage collectors may be classified in two base categories: “reference counting” and “tracing”.

Reference counting garbage collectors keep a count of how many times an object is referenced. Every time a new reference to an object is created, the counter is incremented; when a reference is overwritten or goes out of scope, the counter is decremented. When the counter reaches zero, the object is considered to be unreferenced and is destroyed.

A problem with reference-counting garbage collectors is circular references – if two objects contain a reference to each other, their reference count will never be zero – thus, a cycle detector is required to complement a reference counting garbage collector.

Since D is not a managed language, it cannot track object references, and thus a reference counting garbage collector is not compatible with D's design. This leaves out tracing garbage collectors.

Tracing GCs are so called because they “trace” through the working set of memory. They determine which objects are “reachable” by following references from static memory and thread stacks, and then discard the rest. An example is illustrated in Figure 1.

“Reachability” can be described as follows:

a) There is a distinguished set of objects called “roots”, which are usually located outside the managed memory heap. In practice these usually constitute the program’s static data segment (global variables) and the stacks of all running threads (local variables). These objects are considered to be initially reachable.
b) Anything referenced from a reachable object is considered reachable (we can say that reachability is a transitive closure).

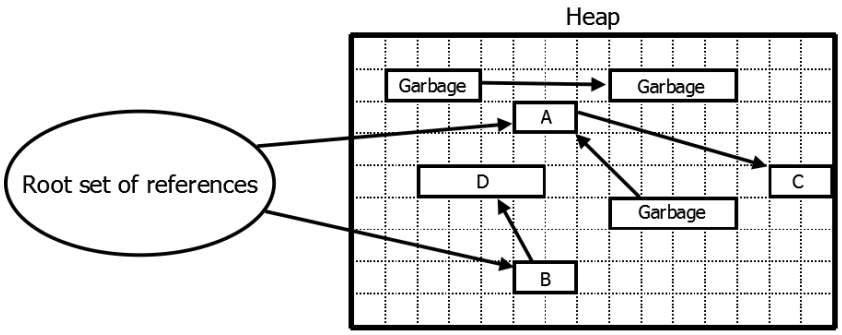


Figure 1 – reference tracking

Tracing garbage collectors perform collection in “cycles”. A cycle comprises a GC run, and occurs when the collector decides to reclaim memory, usually when the program is low on memory.

Tracing garbage collectors can further be classified by their basic algorithm:

a) Mark-and-sweep – in the mark-and-sweep method, all objects are divided in two sets, black and white (reachable and possibly-reachable). When a collection cycle starts, the root objects are added to the black set, and the rest to the white one. Iteratively, the collector scans any references from the black set, and moves referenced objects from the white set to the black set. When there are no more references from the black set to the white set, the objects in the white set can be safely discarded. The disadvantage of the mark-and-sweep garbage collector is that it is a “stop-the-world” collector, which means all threads must stop during the collection cycle.

b) Tri-color marking – a tri-color marking garbage collector uses three sets instead of two. The three sets are white (condemned), gray and black. Initially, the gray set contains the root objects. The collector then iteratively processes every object in the gray set by “blackening” the object (moving it to the black set) and “graying” all objects referenced by the object in the white set. This cycle is repeated until the gray set becomes empty. At the end of the cycle, all objects in the white set are provably not reachable and can be reclaimed. The advantage of the tri-color marking is that for managed languages it does not require to stop all threads from execution, because new object references can simply update the three sets in real-time, during a

collection cycle – thus allowing “on-the-fly” garbage collection (in contrast with the previous one). Unfortunately, this technique requires the language to be able to track the creation of object references, which is not possible with a systems programming language, and thus is not suitable for the D programming language.

Garbage collectors can also be classified by the various implementation strategies:

a) Moving vs. non-moving: a “moving” garbage collector is different from a “non-moving” garbage collector in that it moves objects in memory, updating all references to these objects. This design allows it to perform “heap compaction” – a technique which defragments the memory by moving all objects into a minimum address range (see Figure 2). Although a moving GC may seem inefficient, it has several benefits, such as quick allocation (unallocated space is continuous) and quick deallocation during a collection cycle (an entire continuous region is marked as free, instead of deallocating individual objects).

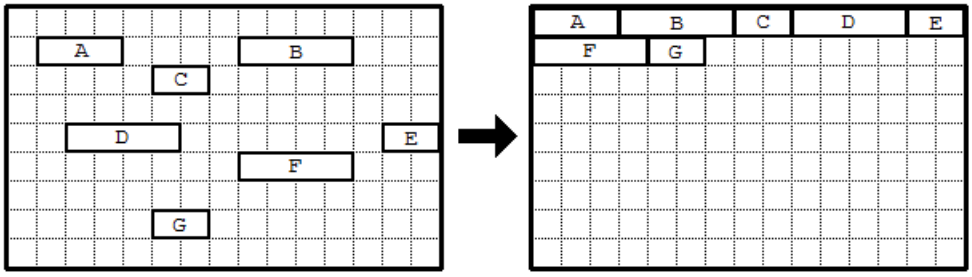


Figure 2 – heap compaction

Additionally, if appropriate traversal order is used, objects that refer to each other frequently can be moved very close to each other in memory, thus increasing the likelihood that they will be located in the same cache line or virtual memory page. This can significantly speed up access to these objects through these references. However, since unmanaged languages like C/C++/D do not track object references, a moving GC is not possible to implement without precise data type information from the compiler: in order to adjust references to moved objects, the GC requires a precise memory map indicating which memory fields are pointers, and which are data.

b) Copying vs. mark-and-sweep vs. mark-and-don’t sweep: a “copying” garbage collector (also called a “stop-and-copy” or a “semi-space” collector) partitions all memory into two regions: “from space” and “to space”. Initially, objects are allocated into “to space” until they become full and a collection is triggered. At the start of a collection, the “to space” becomes the “from space”, and vice versa. The objects reachable from the root set are copied from the “from space” to the “to space”. These objects are scanned in turn, and all objects that they point to are copied to “to space”, until all reachable objects have been copied to “to space”. Once the program continues execution, new objects are once again allocated from the "to space" until it is once again full and the process is repeated. The advantage of a copying GC is in its simplicity, but the disadvantage is large memory requirements during a GC cycle. This technique has been used as early as 1969.

   A “mark-and-sweep” garbage collector maintains a bit for each object tracked to record whether it is “white” or “black”. As the reference tree is traversed during a collection cycle (the “mark” phase), the bits are set to reflect the current state. A final “sweep” of the object set then deallocates “white” objects. A “mark-and-don’t-sweep” collector is a variation of the “mark-and-sweep” collector – the difference constitutes in that all objects are categorized in “white” and “black” sets. The “white” set represents unreferenced objects. When a memory allocation is requested, the collector first tries to find an appropriate “white” object to reuse. When there are no more “white” objects, the collector scans all objects for references and rebuilds the “white” set with unreferenced objects.

c) Generational GCs – also known as “ephemeral GCs”, these garbage collector designs are based on the observation that in many programs, most recently-created objects are very short-lived (are likely to become unreachable quickly). A generational GC divides objects into generations and, in most cycles, will only scan for references to objects in the more recent generations. The set of the most recent generation of objects is sometimes referred to as “nursery”. Furthermore, the framework tracks object references and is notified when object references cross generations. When a collection cycle runs, it may be able to use this information to

prove that some objects in the recent generations are unreachable without having to scan the entire memory.

d) Stop-the-world vs. incremental vs. concurrent – a “stop-the-world” garbage collector design requires that the program completely stops execution while a collection is performed – namely, all other threads must be paused. The obvious disadvantage of such designs is that programs run with sporadic delays, as they must periodically stop execution to reclaim memory. Incremental garbage collectors perform collection in small increments during the execution of the program, thus causing no noticeable delays. Concurrent collectors run in a separate thread, and do not stop other threads at all. Incremental and concurrent collectors require a very careful design to avoid race conditions disrupting the state of the collection, and thus allowing for an object to be erroneously marked as unreferenced.

e) Precise vs. conservative – a “precise” (also “accurate” or “exact”) garbage collector tracks information about all object references, and knows precisely which objects hold references to other objects. This allows it to precisely identify whether an object has any references to it or not. In order to be able to do this, precise garbage collectors require the framework to either track all object references, or maintain precise type information about the system. The disadvantages of precise GCs are that they are slower than conservative GCs and complicate the design of the compiler or framework. In contrast, a “conservative” garbage collector does not differentiate from pointers and other data, and instead scans all allocated memory for bit patterns which, if interpreted as pointers, point inside the managed memory heap. Conservative garbage collectors are used for compiled languages, such as C/C++/D, where casts may be used to cast pointer objects to other types. The disadvantage of conservative garbage collectors is that they may not reclaim an unreferenced object simply due to the fact that there happens to be a bit pattern of data which, if interpreted as a pointer, would point inside the object.

# 2 D Programming Language

## 2.1 Overview

The D programming language, designed and implemented by Walter Bright, appeared in 1999 when W. Bright decided that too many mistakes have been made in the design of C++, and took upon himself to create a new programming language to address C++'s shortcomings while, at the same time, adding features from modern managed programming languages. Walter Bright was the main developer of the first native C++ compiler, Zortech C++ (later to become Symantec C++, now Digital Mars C++), and thus already had experience with writing compilers. Since the first version released in 2001, the language was in continuous development, and is continuously attracting the attention of more developers.

D is a "systems" programming language, meaning it can be used for low-level task such as writing device drivers and operating systems (indeed, there is a project for writing an operating system entirely in D). However, this does not imply the relative inconvenience of use usually associated with compiled languages such as C (when comparing to C#/Java or interpreted languages). On the contrary, D attempts to combine the power of high performance of C and C++ with the programmer productivity of modern languages like Ruby and Python. D also concentrates on quality assurance, documentation, management, portability and reliability.

The D programming language is statically typed and compiles directly to machine code. It is not concentrated on a single paradigm, and supports many programming styles: imperative (procedural), object oriented, and meta-programming. Recent additions to the language also allow programs to be written in a functional style. Since it is not associated with a company or commercial organization, it is not governed by a corporate agenda and is not being designed by a committee – most of the design is influenced by the success of other languages and is "community-driven" (the community's feedback greatly affects the language's design).

D has received coverage in press on several occasions. The book “Learn to Tango with D” was written by Kris Bell, Lars Ivar Igesund, Sean Kelly and Michael Parker in 2008, and published by firstPress. A German book “Programming in D: Introduction to the new Programming Language” is available, as well as a Japanese book “D Language Perfect Guide”. Andrei Alexandrescu, a renowned expert on advanced C++ programming, author of the book “Modern C++ Design”, is expected to release a book titled “The D Programming Language” in October 2009.

There are currently several D compiler implementations. The reference implementation, Digital Mars D, is available for Windows, Linux, FreeBSD and OS X. The reference implementation is open-source, with the back-end being under a restrictive open-source license and the front-end and standard library being free software which can be freely reused.

Third-party implementations include GDC (GNU D compiler) and LDC (LLVM D compiler). The GDC compiler adapts the official front-end and standard library with the GNU C compiler (GCC), thus allowing D to be used on virtually any platform to which GCC has been ported. The LLVM D compiler is based on LLVM (“Low Level Virtual Machine”), a new compiler infrastructure designed for compile-time, link-time, run-time and “idle-time” optimization of programs written in arbitrary languages. LDC is considered production-ready for Linux 32-bit and 64-bit platforms, with more platform support on the way as LLVM development continues.

### 2.2 Notable features

The D specification lists numerous new and improved features compared to C++. This section describes several features relevant to the scope of this paper.

#### 2.2.1 Dynamic arrays

Aside from support of C-like static arrays, D introduces built-in *dynamic arrays*. These arrays are different from the general notion of dynamic arrays in other languages.

A dynamic array is declared as follows:

```
int[] a;
```

A D dynamic array is internally represented as a structure with a pointer and length. The pointer indicates the start of the array data; the length indicates the number of elements in the array. These fields can be accessed using the respective properties – `a.ptr` and `a.length`.

D allows the following array operations: allocation, indexing, resizing, slicing, concatenation, appending, duplication, deallocation.

Allocation can be done using the **`new`** keyword, for example:

```
a = new int[1000];
```

This will allocate an array of 1000 integers.

Indexing is done like with regular arrays:

```
int i = a[5];
```

Resizing is done by setting the **`length`** property:

```
a.length = 500;
```

Note that this will not result in reallocation of the array. When the length is set to a value smaller than the previous one, no reallocation occurs – only the length value is updated in the array structure. Reallocation does happen when the length is set to a value larger than the previous one.

Slicing arrays is done using the slicing operator:

```
int[] b = a[100..200];
```

Slicing will create a new array construct, which points to the 100$^{th}$ element of `a` and has a length of 200, but will not reallocate memory. Thus, writing to an element of the `b`

array will also modify the `a` array. This may sound counter-intuitive, however it opens up the path to new programming strategies, allowing to write more efficient code, because it allows programmers to avoid memory duplication. Omitting the contents of the square brackets is valid, and indicates a slice of the entire array (which is usually used to create a dynamic array slice over the data in a static array).

Unlike many other languages, D has a dedicated operator for concatenation, ~. Two arrays are concatenated as follows:

```
int[] c = a ~ b;
```

Unlike slicing or resizing, concatenation will always result in memory allocation. Since D is a garbage-collected language, programmers are free to use slicing and concatenation at will without having to worry about deallocating unused array contents. Similarly, an object can maintain a reference to an array or a slice of it for an indefinite time, without having to worry that the component which originally allocated the array would deallocate it, leaving a dangling pointer.

Array appending is similar to concatenation:

```
c ~= b;
```

Although intuitively similar to `c = c ~ b`, it is different in that it will not always allocate memory. When appending to an array, the garbage collector can check if there is enough free memory after the end of the array, and extend the array in-place. Similarly, the GC may decide to extend the array by a size larger than the item or array being appended, to optimize further appending operations.

Duplication simply creates a copy of an array in memory, and is done using the **`.dup`** property:

```
int[] d = c.dup;
```

Explicit deallocation of arrays is not necessary

A class of array operations unrelated to memory management but still worth noting is *vector operations*. A vector operation is indicated by the slice operator appearing on the left of the assignment operator (or a combined assignment/operation operator, such as +=). Vector operations allow the compiler to apply platform-specific optimizations, such as generating MMX/SIMD processor operands. For example:

```
d[] = c[] + 5;
```

This will copy in `d` all elements in the `c` array, added with 5.

The D language does not have a distinct “string” data type. Instead, it uses character arrays to represent strings. String literals, such as those in C/C++, are represented internally as static arrays of characters. The “string” alias This makes redundant several string functions present in other languages used for operations such as concatenation or substring slicing.

### 2.2.2 Associative arrays

Associative arrays, sometimes also called “hashtables” due to the use of hash functions in most implementations, represent a mapping from objects of one type to another. It is used to associate certain information with values that do not form a contiguous range – for example, given a list of students, one can associate their group name with the student name.

The advantage of associated arrays is that both insertion and lookup are done very quickly, and do not require iterating over every element to find the needed one. D has built-in support for associative arrays. The syntax is as follows:

```
int[string] aa;
```

The type before the brackets is the type of the value; the type inside the brackets is the type of the key.

Adding a value to an associative array is done with the following syntax:

```
aa[“apples”] = 5;
```

Following this, the value can be accessed using the expression `aa[“apples”]`.

Elements can be removed from the associative array using the `.remove` method.

Associative arrays rely heavily on D’s automatic memory management. The language specification does not specify any means to completely delete memory allocated by associative arrays.

### 2.3 D Garbage Collector

#### 2.3.1 Overview

The standard D garbage collector is based on Hans Boehm's C++ Garbage Collector. The basic idea is to handle memory allocations using preallocated memory pools, and when memory runs low to recursively scan memory, starting with root objects, for pointers to other objects.

The source code of D's default garbage collector can be found in the dmd\src\phobos\internal\gc subdirectory of the distribution ZIP file. The code is distributed among several modules, which are described below:

a) gc.d – implements an interface between the compiler (namely, functions calls to which the compiler generates) and the garbage collector implementation.
b) gcbits.d – contains a helper structure (GCBits) which manages an array of bits.
c) gcx.d – the actual GC implementation, contains most of the code.
d) gcold.d – compatibility code for older libraries.
e) gclinux.d, win32.d, gcosx.c – platform-specific code.

#### 2.3.2 Memory layout

The GC stores objects inside large contiguous memory spans allocated by the OS, called "pools". Pools are allocated with a size multiple of 64kB. Initially, most memory in a new pool is marked as "reserved" (mapped, also referred to as "virtual memory") and does not represent physical memory. As more memory is required, the GC will request the OS to map memory to the reserved address spans.

Memory is further subdivided into "pages". Each page is 4kB in size. Pages can be further subdivided into "bins", which represent a cell for storing smaller objects. A page's "bin size" represents the size of the objects allocated in the page. Bins start at 16 bytes, doubling their capacity up to the size of the page. Pages are categorized by the size of their bins – for example, a page with 256-byte bins can store 16 objects (typically between 129 and 256 bytes). Objects larger than the size of a page are stored with their first 4kB in a page with a "page" bin size, and the rest in a contiguous range of pages with "page plus" bin sizes.

The garbage collector maintains several flags regarding each allocated object. The granularity of the flags is equal to the size of the smallest bin (16 bytes):

a) “mark” – set during a collection cycle after an object has been scanned.
b) “scan” – set during a collection cycle for an object that is yet to be scanned.
c) “free” – set when the object is free (applies to objects inside bins).
d) “final” – set when the object requires a finalizer to be called when the object is deallocated.
e) “noscan” – set when the user or the runtime specified that the object shouldn’t be scanned for pointers.

Figure 3 is an example illustration of the state of the memory at one point in a program’s execution.

```
Page map for pool 022B0000 - 023B0000 ( 256/ 256/ 256 pages):
(page size = 0x1000)            +10000           +20000           +30000
022B0000: 694579015P+890P+ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
022F0000: ++++++++++++++++9 5017119115167869 1511187111511817 1511811157811418
02330000: 1915791891519874 9995891918759698 6479159116911691 9861519941611199
02370000: 1689161591971661 9948115196916111 9164111911619171 1691114517916111
Page map for pool 02730000 - 029E8000 ( 627/ 640/ 696 pages):
(page size = 0x1000)            +10000           +20000           +30000
02730000: P++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
02770000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
027B0000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
027F0000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
02830000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
02870000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
028B0000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
028F0000: ++++++++++++911 6195171169184191 6711519618167669 P+++++65661656P6
02930000: 66767076757677 67 P776776P++++P+++ +P++++81917614P+ +++P++++91P++++P
02970000: ++++1161PPPPPP51 P++++++P++++++98 17611P++++++P+++ +++.............
029B0000: xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxx
```

Figure 3 – example memory map

Each character in the memory map represents a page. The character itself represents the type of the page:

a) digits 4,5,6,7,8,9,0,1 represent the size of the bin – 16,32,64,128,256,512,1024,2048 bytes respectively (the digit represents the last digit from the power of 2 of the corresponding size);
b) “P” represents a “page” bin (a page containing an object between 2049 and 4096 bytes, or the first 4096 bytes of a larger object);
c) “+” represents a “page plus” bin (containing the continuation of large objects);

d) “.” represents unallocated space;

e) “x” represents reserved (uncommitted) space.

The varying color intensity of the map elements represents whether or not the GC will scan said areas for pointers. Dim characters represent objects marked as not having pointers, while enunciated characters represent objects marked as possibly having pointers.

Additionally, the GC also maintains “free lists”, which represent unallocated objects inside pages. A free list exists for every bin size from the smallest (16 bytes) to 2048 bytes.

### 2.3.3 Memory operations

Most memory operations are synchronized, meaning only one can be executed at the same time. This is done to prevent several threads from modifying the GC structures simultaneously, thus corrupting memory.

New memory is allocated differently depending on the size of the object. Objects larger than 2048 bytes are allocated in whole pages. If there are no free pages available in memory, the GC performs a collect cycle. If the collect cycle didn’t free enough memory, a new pool is allocated. Pools are allocated with an ever-increasing size, to better optimize the performance of memory-intensive applications.

Smaller objects are allocated similarly, however the GC first checks the free list for the corresponding bin size. If the free list isn’t empty, it simply reuses the object in the free list; otherwise, it tries to allocate a new page as described above.

Freeing objects is straight-forward: for small objects, they are added to the free list; for large ones, their pages are marked as free outright. Finally, the destructor is called on the object, if required.

Memory reallocations are implemented as follows: if the object is allocated as one or several pages, and is being shrunk to a size still at least fitting a page (lager than 2048 bytes), then it is shrunk in place. Page-size objects are also attempted to be expanded in-place – assuming the memory after the object is free. Otherwise, new memory is allocated and the object is copied over.

### 2.3.4 Garbage collection

Garbage collection is done in the following procedure:

a) All threads other than the current thread are paused.
b) The “mark”, “scan” and “free” flags for all pools are cleared.
c) For every object in the free lists, its “free” flag is set, so that it would not be scanned.
d) The “free” flags for all pools are copied over the “mark” flags. This is an optimization, so the GC won’t have to check the “free” bits as well as the “mark” flags when scanning.
e) The root set of objects (static data and the stacks of all threads) is “marked”. Marking is described below.
f) As long as any changes are made to the pools’ flags, the following cycle is executed: for every object with the “scan” flag set, its “scan” bit is cleared and the object is “marked”.
g) Every object’s “mark” flag is checked. If it isn’t set, then the object is freed (its destructor is called, if the “final” flag is set). Objects smaller than “page size” are not added to the free list at this phase.
h) Free lists are rebuilt, and pages containing entirely free bins are freed.
i) All threads are again resumed.

“Marking” is a procedure which scans a contiguous range of memory (typically holding a single object) for pointers towards other objects. Specifically, it searches for pointer-sized values in the scanned range (4 bytes on 32-bit systems, 8 bytes on 64-bit systems) which – when interpreted as a pointer – point towards an object managed by the GC. For each of these objects whose “mark” flag is cleared, the “mark” flag is set and, if the object’s “noscan” flag isn’t set, its “scan” bit is set as well. This will ensure correct propagation of the “mark” and “scan” bits across all references in the GC’s memory heap.

# 3 Problems and solutions

As described earlier, garbage collection is not a panacea, and does not automatically resolve all memory-related problems. In fact, depending on the design of the garbage collector, it may even introduce new problems which programmers need to be aware of.

During the development of several projects written in the D programming language, several of these problems were encountered. This chapter describes the methods used to analyze these problems and devise a solution. Some solutions required changes to the implementation of the D compilers and runtime, and other required writing a separate framework to help application developers quickly identify and fix problems in their applications.

## 3.1 Performance

Writing a fast garbage collector is a very complicated problem. Since the memory allocation requirements vary drastically from one application to another, it’s hard to create a GC which runs equally well on all applications. However, it is sometimes possible to find an optimization which drastically improves a certain common use case, without severely affecting the overall performance in other cases.

One such example of a program was posted on the digitalmars.D forum by user “bearophile”:

```
import std.file, std.string;
import std.stdio;
static import std.c.time;
double clock() {
    auto t = std.c.time.clock();
    return t/cast(double)std.c.time.CLOCKS_PER_SEC;
}
void main() {
  auto t0 = clock();
  auto txt = cast(string)read("text.txt");
  auto t1 = clock();
  auto words = txt.split();
  auto t2 = clock();
  writefln("loading time: ", t1 - t0);
  writefln("splitting time: ", t2 - t1);
}
```

Listing 1

This program represents a benchmark for loading a text file and splitting it by words (creating an array of string slices, where each slice represents a word from the input file).

Splitting a large block of text by whitespace will cause an array that's comparable to the size of the original text (a slice is 8 bytes on a 32-bit platform, so if words are 6-7 characters long by average, the size would be the same). Analysis has indicated that the slowdown appeared because the GC needed to look up every one of those pointers, which point to the same large memory region. Thus, for an 8MB text, it had to perform over 1 million look-ups for every collection cycle.

It was noticed that this particular case can be optimized if the GC would cache the result of the previous lookup. A patch for the D garbage collector was written, which would skip scanning a pointer if it pointed inside the same memory page as the previously scanned pointer. The changes can be found in annex A.

The performance improvements introduced by the modification were drastic: the test program's execution time was reduced nearly tenfold, and is illustrated in Figure 4.

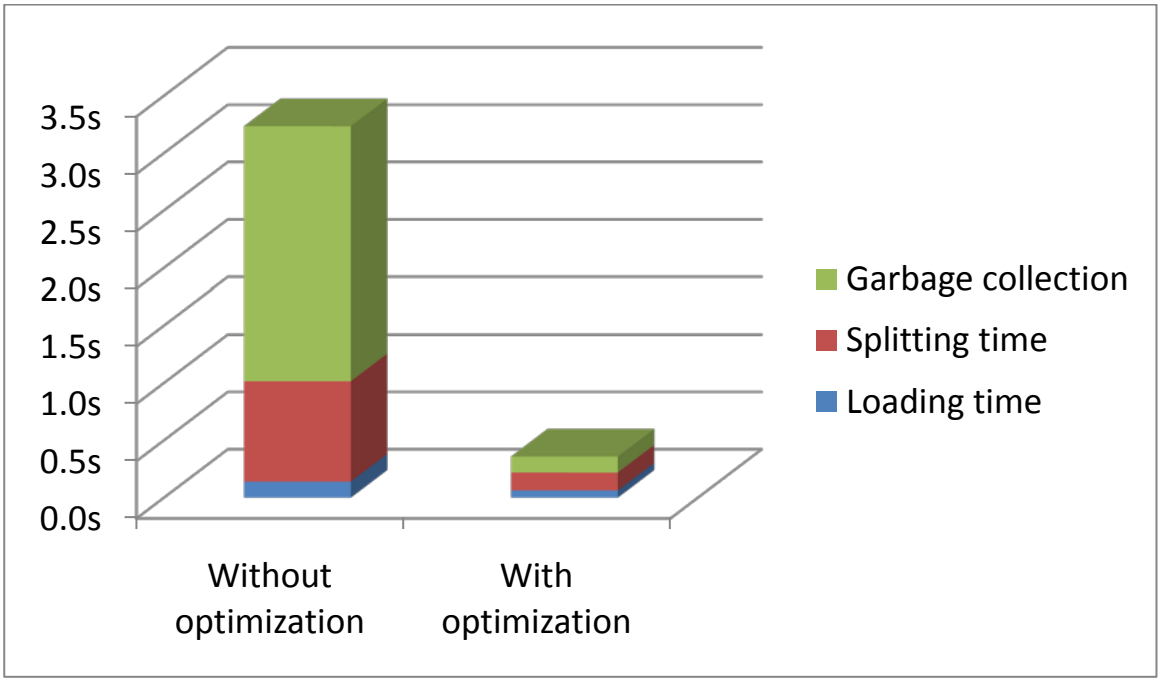


Figure 4 – performance comparison

The patch was published to the digitalmars.D newsgroup and an issue was filed in D's issue tracker. Later, the patch was accepted and became part of the standard D garbage collector, and is currently included in Digital Mars D versions starting with 1.042 [1].

### 3.2 Memory leaks

#### 3.2.1 Description

Considering that D is a garbage-collected language, it may appear that memory leaks are impossible. However, this is not true – the truth being, memory leaks may still occur, however these memory leaks are of a completely different nature than the manual-memory-management ones.

To understand this class of memory leaks, we need to understand how D's garbage collector works. As described in section 2.3, reference tracking towards objects is done by scanning objects for structures which, when interpreted as pointers, point towards other objects.

The problem with this design is that if a data structure, interpreted as a pointer, happens to point inside another objects (which has no other references towards it), that object will not be freed.

This problem was partially addressed when version 1.001 of the Digital Mars D compiler introduced a new "type-aware" GC. What this basically meant was that for each object, the GC maintains an extra flag, which indicates whether the said object "might contain pointers", or "does not contain pointers". This flag is called "noscan" in the implementation, and it is set during object allocation, depending on the type being allocated. Although this change prevents the GC from scanning POD (plain-old-data) objects, which don't contain pointers, it will not affect scanning objects containing mixed reference and data types.

The worst case of a memory leak problem in the context of a garbage-collected language occurs when a large block of memory of sufficiently high information entropy is erroneously marked as containing pointers. The larger the block is, the larger the probability that a pointer-sized byte sequence within the block could be interpreted as a pointer towards existing memory objects.

It is trivial to calculate the probability of this happening:

Suppose we have two blocks of memory, M and N. Block M is a block with random data which is erroneously marked as having pointers, while block N is a block which

shouldn't have any pointers towards it – and would normally be considered as "unreferenced" by the GC.

The chance that a random 32-bit value, interpreted as a pointer, will point inside N is N's size divided by the total size of the 32-bit address space, $2^{32}$ – or rather, we can say that the probability of it not pointing inside N will be 1-($N_{size}$ / $2^{32}$). To calculate the probability that no pointer inside M points inside N, we raise that to the power $M_{size}/4$ (since the GC only scans values aligned at 4-byte boundaries). Thus, the probability that there exists a pointer in M which points inside N is as follows:

$$P(M,N) = 1 - (1 - \frac{N_{size}}{2^{32}})^{\frac{M_{size}}{4}}$$

For values already as small as 1 MB ($2^{20}$ bytes) for M and N, it's almost guaranteed that M will contain pointers pointing inside any N-sized memory block. This is so-called the "minefield" effect (due to the similarity of how bogus pointers create false references and the way mines are spread on a minefield). Thus, programmers have to be very careful when dealing with such pseudo-random data – otherwise, the GC will incorrectly detect pointers to most large objects allocated by the application.

Even though it appears that this situation should almost never appear in practice, a design decision allows this problem to appear frequently. For example, consider the following simple program:

```
import std.file;
void main()
{
	auto data = read("file1") ~ read("file2");
	// (perform some processing)
}
```

This program loads the files named "file1" and "file2" from disk, and concatenates their contents in memory. However, the concatenation instruction causes the garbage collector to allocate a large block of memory marked as containing pointers. If the files are large and have a high-enough entropy, then as long as a root reference towards these files is maintained, most large objects allocated by the program will never be deallocated.

This happens because the “read” function, which loads a file from disk returns its contents in memory, returns a void[]. A void[] (void array) is the dynamic array counterpart to the void pointer type (void*). The concept may seem unintuitive, however it is simply a void pointer and a length field (see section 2.2.1).

The problem stems from the fact that void[] is considered as a type which may contain pointers. The logic behind this decision is that a void[] may contain absolutely anything – including pointers. If this behavior would be altered - should a programmer store the only pointer to an object inside a void[], the GC would not find the reference to the object and free it.

Although the read() function explicitly specifies that the memory allocated for the file’s contents is not to be scanned for pointers, the concatenation operation will allocate a new block of memory with the properties of a void[] – including the “has pointers” property. This causes the GC to scan the file contents as if it contained pointers to other objects, thus detecting spurious object references.

### 3.2.2 Resolution

The first encounter with memory-related problems while using D was during the development of a scanner application as part of the author’s employment at WebSafety Inc. The problem was that the scanner was using an unusually large amount of memory – far more than it should use at any given time during execution.

Since the problem was with a commercial application, finding a solution to the problem was essential. After several failed attempts of trying to track down the memory leak using “traditional” debugging methods, it was decided to create the first memory debugger and profiler for the D programming language.

The memory debugger will be fully described in the next chapter. Its feature which helped me in this case was disk logging of the program’s memory, and post-mortem analysis which allowed finding all references to a block of memory. This allowed to quickly identify which object contains references to an object that should have been freed, as well as identify any blocks of memory erroneously marked as containing pointers.

### 3.3 Memory corruption

Memory corruption is a class of bugs that software developers fear most. The reason for that is that the cause and manifestation of a memory corruption problem can be very different. Memory corruption is often very hard to consistently reproduce, which further increase the effort required to locate the cause of the problem. For these reasons, a language platform which allows programmers to operate with memory directly (e.g. using pointer operations) cannot be complete without good memory corruption debugging tools.

D is designed in such a way as to minimize the required use of pointers, and indeed – large D applications can be written without using pointers at all. This is mostly due to D's dynamic arrays and that D's classes are reference types, unlike C++ classes. Although this reduces the odds of memory corruption bugs, these may still occur. Of course, programmers requiring every last bit of performance will likely continue to use lower-level code constructs involving pointers and manual memory management, thus exposing their code to these categories of problems.

The memory debugger described in this paper can catch two types of memory corruption bugs: dangling pointers and double free bugs.

#### 3.3.1 Dangling pointers

A frequent case of memory corruption occurs when an object is accessed shortly after it has been deallocated. Since the actual memory is not cleared when the object is destroyed, it is still possible to access the object's data through the old object – that is, until a new object is allocated at the same address. Since the actual behavior of the memory manager can be unpredictable, this can cause bugs which only don't work in specific cases, and may stop manifesting themselves when the programmer attempts to study it (commonly named "heisenbugs" [4]).

The debugger allows programmers to quickly catch dangling pointer bugs using a technique called "memory stomping". Basically, it simply involves overwriting deallocated memory with garbage. Thus, any code which relied on an object's data still being at the same address after the object was freed will fail immediately, allowing developers to spot the problem much faster.

The following listing is a test case for the memory stomping feature:

```
import diamond;

void main()
{
     int[] z = new int[5];
     int* i = &z[2];
     *i = 5;
     delete z;
     assert(*i != 5);
}
```

In this test program, i is the dangling pointer inside the z array. After z is explicitly deallocated, the memory pointed at by i will no longer contain the values which used to be in z.

### 3.3.2 Double free bugs

A “double free bug” happens when the program attempts to deallocate a memory region which is already free. If another object happens to have been allocated at that address between the two deallocations, that object will be freed instead. Depending on the memory management code, double free bugs may lead to undefined behavior and memory corruption.

The memory debugger can detect double free bugs by explicitly checking if there is actually an object allocated at the specified address. If it isn’t, the program will crash instantly, allowing the developers to immediately locate the second free operation.

The following program is a simple example of a double-free bug which the debugger intercepts:

```
import diamond;

void main()
{
     auto a = new ubyte[10];
     auto b = a;
     delete a;
     delete b;
}
```

# 4 Diamond Memory Debugger

## 4.1 Overview

"Diamond", the D Memory Debugger, is composed of two components: a module and a post-mortem memory log analyzer.

The module contains code which is compiled as part of debuggee programs. This module intercepts calls to the standard G garbage collector and other memory management functions, and performs tasks such as verification and logging.

The post-mortem memory log analyzer is an interactive console application which allows developers to inspect memory logs generated by the module. It can be used to detect causes of memory leaks, memory corruption and generally to inspect the contents of the program's memory along the program's execution.

## 4.2 Module

### 4.2.1 Usage

In order to use Diamond, a programmer needs to import the "diamond" module at the start of their program, as the first imported module. This is so that the module could initialize as early as possible, thus applying its effects to any memory operations performed during the initialization of other modules.

The behavior of the module is configurable via several options at the top of the module's source. The programmer is required to edit these options to adjust the module's behavior according to their requirements. The available options are:

a) MEMSTOMP – enable "memory stomping", which is used to detect dangling pointer bugs and some other forms of memory corruption.
b) FREECHECK – enable verifications on free operations, which prevent "double free" bugs.
c) MEMLOG – enable writing a memory log (containing a full memory map and the memory contents) for every garbage collection cycle.
d) MEMLOG_VERBOSE – enable writing memory logs every few MEMLOG_VERBOSE_STEP memory operations. This option slows down the program considerably, but generates the most verbose memory logs.

e) MEMLOG_VERBOSE_STEP – when MEMLOG_VERBOSE is enabled, this option specifies how often should a full memory dump be written. The default value of 1 creates a memory dump before and after every memory operation, while a value of 10 would cause memory dumps to be written on every $10^{th}$ memory operation.

f) MEMLOG_CRC32 – enables memory log compression by caching memory pages using their CRC32 values. This option increases CPU usage, but greatly decreases the size of the generated memory logs.

g) LOGDIR – path prefix for generated memory logs (specifies directory where memory logs will be saved to). The default (empty) value causes memory logs to be saved to the current folder.

### 4.2.2 Implementation

The module imports the modules containing the implementation of D's garbage collector. These modules are not normally accessible to applications, so some configuration may be required. This is required to get access to the definition of the garbage collector class.

The module defines a class type which is derived from the class containing the standard D implementation. This class implements a proxy pattern – it is designed to intercept memory operations, perform any required debugging functions, then pass them to the standard GC. However, this class is never instantiated – instead, during initialization, its virtual function call table is used to overwrite the virtual function call table of the original garbage collector class, thus rerouting all virtual method calls to our class type. Practically, this means that the type of the garbage collector instance is being changed at runtime.

However, the GC class does not contain all methods that required to be intercepted. Several more functions are intercepted using a technique called "code hooking". This involves patching the executable code at the function's address to a jump to our function. This technique requires editing memory access flags to prevent an access violation or segmentation fault, which is platform-dependant.

It should be noted that the module is written to work with both of D's major standard libraries, Phobos (the standard one) and Tango (a very popular 3rd-party standard library). The implementation details of memory management and some other aspects vary greatly between these two libraries, so the module contains two versions of some code.

### 4.2.3 Logging

The module's most significant feature is its ability to perform a detailed memory log of every memory operation. Every memory allocation, reallocation, deallocation, as well as garbage collection cycles are logged to a binary memory log. All events are logged with a timestamp and a full stack trace.

Garbage collection events are handled differently by the module. Before the garbage collect, the module will save the complete state of the application's memory, including the entire content of the heap. After the collection cycle, the module logs the memory map. This allows the programmer to visualize which objects were freed during a collection cycle, as well as allow the analyzer to maintain the memory state of the application.

In cases where detailed memory inspection is required, the programmer may enable verbose memory logging. This will cause the module to save memory dumps every N memory operations, where N is the value of the MEMLOG_VERBOSE_STEP configuration option. Since using this option could generate very large memory maps, the MEMLOG_CRC32 option was added.

MEMLOG_CRC32 will store the CRC32 value of every page at the moment when it was last saved to the log file. Thus, if a page has not been modified since the last memory dump, then it is not written a second time. Even though this option increases CPU usage (since the CRC32 values for every page must be calculated), it can greatly reduce the size of the generated log files.

Additionally to the standard logging functions, the module also exposes some logging features available to the debugged program. For example, the program may call the logText() function to log a text string (which will be displayed in the log analyzer). Similarly, the program may call the logMemoryDump function to trigger a full memory dump at any point during execution.

### 4.3 Analyzer

#### 4.3.1 Overview

The post-mortem memory log analyzer is an interactive console application which allows developers to examine the contents of memory dump files in a variety of ways, to help identify problems such as memory leaks. It is a versatile tool with a wide variety of commands that can be used to track the application's memory usage and contents along the logged run.

The program takes two optional command-line arguments – the memory log to be analyzed, and the application map file. If either of these are not specified, the analyzer tries to load the most recent .mem and .map file from the current directory. The map file specifies the addresses of symbols (such as functions and variables), and is necessary to display function names in stack traces, but is not required.

User interaction is provided by the means of a command-line prompt. The prompt line consists of a single character representing the type of the current event, followed by the number of the event. When the analyzer is initially loaded, it is located at the start of the file, and is not pointing at any event.

The "current event", also called the cursor, is the event currently being selected. Since events are written to the log in chronological order, the event number increases with the program's execution time. Some commands only affect the current event. The "goto" and other commands can be used to move the cursor to another event number.

During analysis, the analyzer maintains an internal memory map of the application. This allows the user to cross-reference events at certain addresses, e.g. to find out which event is responsible for allocating memory at a certain address.

Figure 5 represents a typical analyzer session for determining the source of a memory leak. In this case, the problem was a 1.8 MB memory allocation at address 027F0000. The call stack at the bottom shows the exact sequence of functions called to allocate said memory block.

```
Diamond Memory Log Analyzer, v0.1
by Vladimir "CyberShadow" Panteleev, 2008-2009

Using the most recent memory dump file.
Loading diamond_2009-05-06_12.03.38.mem...
Error: not enough data in stream
97758 events loaded.
Using the most recent map file.
3251 symbols loaded from HostingBuddy.map.
> dumps
    426 @ 12:03:38: MEMDUMP (    13/    16/   256)
   4471 @ 12:04:39: MEMDUMP (   618/   624/   952)
   9092 @ 12:08:39: MEMDUMP (   708/   720/   952)
  28607 @ 12:26:05: MEMDUMP (  1464/  1464/  1464)
  28631 @ 12:26:07: MEMDUMP (  1071/  1464/  1464)
  29050 @ 12:26:16: MEMDUMP (  1923/  2232/  2232)
  32146 @ 12:27:14: MEMDUMP (  2148/  2232/  2232)
  50359 @ 12:45:36: MEMDUMP (  2232/  2232/  2232)
  65203 @ 13:00:39: MEMDUMP (  2135/  2232/  2232)
  80789 @ 13:08:40: MEMDUMP (  2130/  2232/  2232)
  81262 @ 13:08:54: MEMDUMP (  1416/  2232/  2232)
  91157 @ 13:13:30: MEMDUMP (  3159/  3240/  3256)
> goto 4471
D4471> map
Page map for pool 001C0000 - 002C0000 ( 157/ 160/ 256 pages):
(page size = 0x1000)          +10000          +20000          +30000
001C0000: 694579015P+890P+ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
00200000: ++++++++++++++++9 5017699871567806 9569P++879469589 6798569986945796
00240000: 8967598697459689 1679569849167... xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
00280000: xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
Page map for pool 027F0000 - 02AA8000 ( 461/ 464/ 696 pages):
(page size = 0x1000)          +10000          +20000          +30000
027F0000: P+++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
02830000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
02870000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
028B0000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
028F0000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
02930000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
02970000: ++++++++++++++++ ++++++++++++++++ ++++++++++++++++ ++++++++++++++++
029B0000: ++++++++++++...  xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
029F0000: xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
02A30000: xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
02A70000: xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx xxxxxxxx
D4471> eventat 27F0000
    428 @ 12:03:38: MALLOC  027F0000 - 029BC4E9 (1885417 bytes)
D4471> stack 428
 0040265B  void diamond.DiamondGC.mallocHandler(uint, void*) +67
 004029FD  void* diamond.DiamondGC.malloc(uint) +29
 00441D71  __d_arrayappendcT +c9
 00411250  void Schemes.loadDir(char[], bool) +208
 00411516  void Schemes.reloadSchemes() +5e
 0041180C  void Schemes._staticCtor8() +8
 004119F8  __D7Schemes9__modctorFZv +8
 004D70D4  void std.moduleinit._moduleCtor2(class ModuleInfo[], int) +bc
 004D711C  void std.moduleinit._moduleCtor2(class ModuleInfo[], int) +104
 004D711C  void std.moduleinit._moduleCtor2(class ModuleInfo[], int) +104
 00491DFB  __moduleCtor +37
 004376C3  _main +11f
 004F75C9  _mainCRTStartup +a9
 76D0E4A5  __end +7676af95
 779ACFED  __end +77409add
 779AD1FF  __end +77409cef
D4471> _
```

Figure 5 – example Diamond analyzer session

### 4.3.2 Commands

a) General statistics
   1) stats – displays total counts of all events, grouped by event type.
   2) allocstats – analyzes all memory allocations and groups them by the call stack. Displays the top 5 call stacks, sorted by total allocated memory. This command can be extremely useful to quickly determine the code which allocates the most memory.

b) Timeline information
   3) dumps – lists all memory dump events.
   4) maps – lists all memory map events.
   5) events *event* [*event2*] – displays some information about the specified event or events, such as event type, event time, and some event-specific information.

c) Navigation
   6) goto *event* – set cursor at specified event number
   7) next – move forward by one event
   8) prev – move backwards by one event
   9) nextdump/nextmap – move to the next dump/map event
   10) prevdump/prevmap – move to the previous dump/map event
   11) lastdump/lastmap – move to the last dump/map event

d) Address search and cross-reference
   12) eventat *address* [*address2*] – show last event affecting an address/range
   13) alleventsat *address* [*address2*] – show all events affecting an address/range

e) Inspection of a specific event
   14) stack [*event*] – display the call stack for the specified event

f) Inspection of dump/map events
   15) info *address* – show information about a specified address
   16) pools [*event*] – display a list of memory pools
   17) map [*address*|* [*event*]] – display a memory map
   18) refs *address* [*address2*] – search for all references to specified address/range

19) allrefs *address* [*address2*] – same as above, but also search for memory regions which the GC would not normally scan
20) dump *address* [*address2*] – dump memory at specified address

g) Diagnostics
21) integrity – verify the validity of the analysis state
22) freecheck – enable/disable free list checking during analysis

### 4.3.3 Memory map

The layout of the memory map is described in section 2.3.2:

Each character in the memory map represents a page. The character itself represents the type of the page:

a) digits 4,5,6,7,8,9,0,1 represent the size of the bin – 16,32,64,128,256,512,1024,2048 bytes respectively (the digit represents the last digit from the power of 2 of the corresponding size);
b) "P" represents a "page" bin (a page containing an object between 2049 and 4096 bytes, or the first 4096 bytes of a larger object);
c) "+" represents a "page plus" bin (containing the continuation of large objects);
d) "." represents unallocated space;
e) "x" represents reserved (uncommitted) space.

The varying color intensity of the map elements represents whether or not the GC will scan said areas for pointers. Dim characters represent objects marked as not having pointers, while enunciated characters represent objects marked as possibly having pointers.

## 4.4 Solving memory leaks

Solving memory leaks using the Diamond framework is very simple. The procedure is as follows:

a) Edit diamond.d and enable the MEMLOG option
b) Add diamond.d as the first included module in your application
c) Recompile your application and generate a map file
d) Run the application and attempt to reproduce the problem
e) Load the map file and the resulting memory log in the analyzer

f) Jump to the last memory dump (use the “lastdump” command)
g) Print a memory map (use the “map” command)
h) Locate a large object which is marked as having pointers
i) Determine the event number of the event which allocated the object (use the “eventat” command)
j) Print the stack of the said event (using the “stack” command)
k) If the stack trace indicates that this object is expected to contain pointers, repeat from step h)
l) Modify your program to prevent the allocation code shown in the call stack from marking your object as having pointers.

Alternatively, it is possible to use the analyzer to find all the references which prevent an object from being deallocated. This is trivially done using the “refs” command, which will display a list of all references (pointers) towards the specified memory range.

### 4.4 Practical applications

Following the development of the Diamond memory debugger, it has successfully utilized it in several personal and business projects.

#### 4.4.1 WebSafety Scanner

WebSafety Scanner, a commercial product from WebSafety, was the application which originally prompted the development of the Diamond memory debugger. The WebSafety Scanner is a desktop application which scans the local computer’s file system, browser and IM logs for any indecent files or entries.

The early versions of the scanner suffered from a severe memory leak. It would use amounts of over 300MB of memory, something unallowable for desktop applications. An example of a run is displayed in Figure 6.

After the development of the Diamond memory debugger and its application in resolving the memory leaks, the memory usage and execution speed were reduced significantly, as shown in Figure 7.

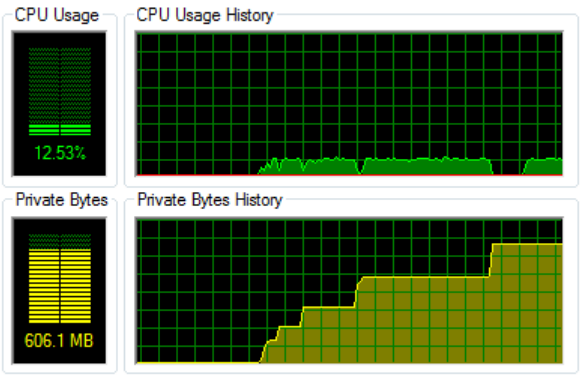


Figure 6 – CPU and memory utilization before the optimization

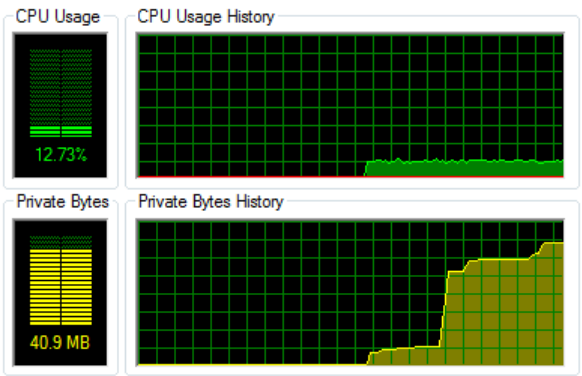


Figure 7 – CPU and memory utilization after the optimization

### 4.4.2 Internet data proxy

The Diamond memory debugger was also successfully applied to a second personal project – an Internet proxy used for routing data via a custom protocol.

The proxy suffered from memory leaks and memory corruption. Using Diamond, both of these problems are resolved. The result is illustrated in Figure 8 – the blue spikes represent the amount of tunneled data, and the red lines indicate times when the proxy had crashed. As you can see, debugging completely eliminated the frequent crashes.

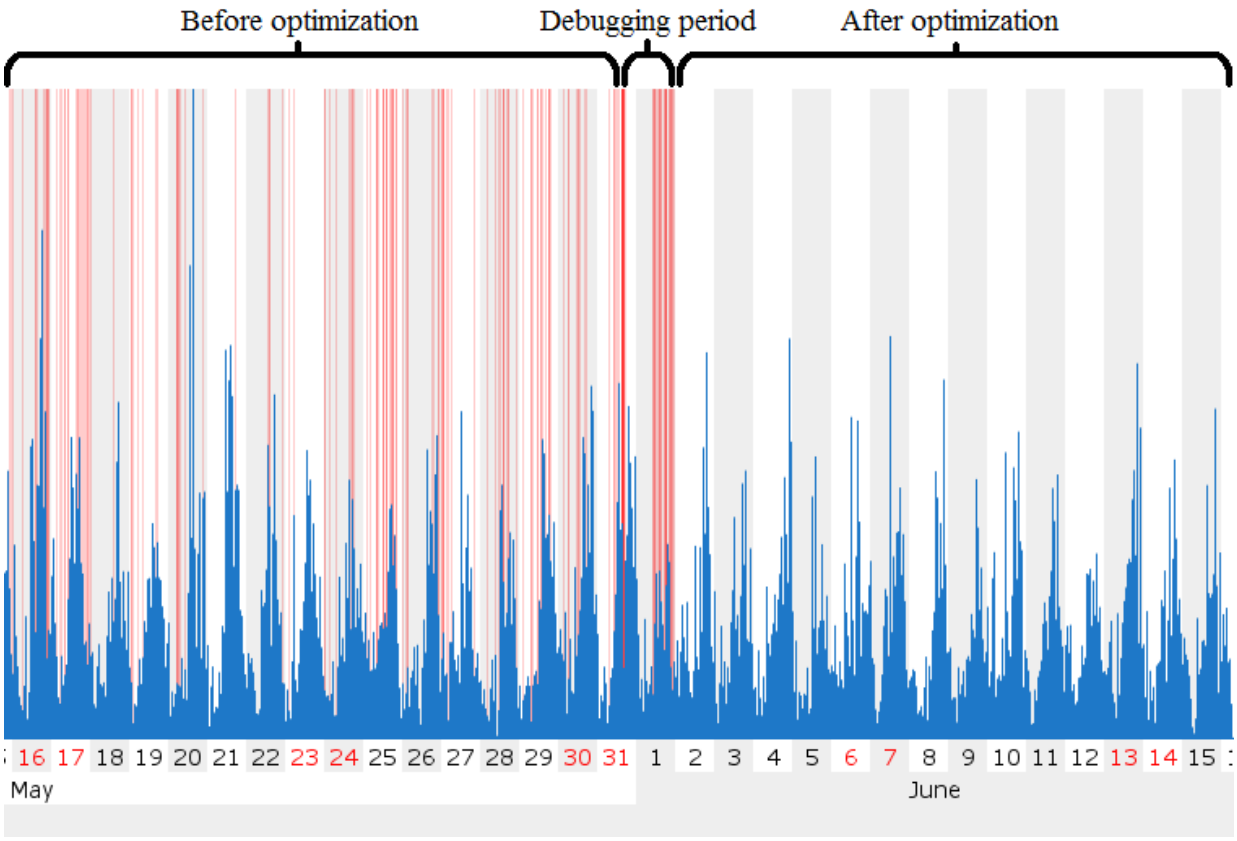


Figure 8 – Data proxy debugging result

# 5. Economic aspects of the project

## 5.1 Description of the project

The evolution of programming languages during the history of computing is directly related to economic aspects of software development. Each new domain-specific language invented is aimed to simplify solving certain problems, while the generic languages build upon the experience of its predecessors to give programmers power and flexibility in solving tasks.

The D programming language is of no exception. Since it was designed by a person with first-hand experience of the software development process in practice, many of its features are aimed at maximizing productivity. A major goal of D is to reduce software development costs by at least 10% by adding in proven productivity-enhancing features, and adjusting language features inherited from other languages so that common, time-consuming bugs are eliminated from the start.

D's other productivity-enhancing goals include:

a) be as portable as possible, despite being a system programming language; all platform-specific elements are abstracted away in the language and runtime library, thus allowing effortless porting to other platforms;
b) have a short learning curve, but adopting the basic syntax and conventions of popular programming languages, such as C/C++, C#, Java;
c) provide direct access to the operating system and hardware when required, thus not having to force programmers waste time writing external "bridge" modules;
d) making it significantly simpler to implement D compilers than C++ compilers, which would allow much faster development time on new platforms;
e) easily support application internationalization, by having built-in Unicode support;
f) incorporate Contract Programming and unit testing features to allow programmers to detect bugs much earlier and test code more efficiently;

g) reduce costs in writing documentation by providing a built-in documentation system (similar to e.g. JavaDoc).

The D programming language is free – the official compilers are free to use and the source code is available for download. The D language itself is not affiliated with a commercial organization, and is not protected by patents or trademarks – thus, anyone may use D and write D compilers without having to ask permission from DigitalMars or pay royalty fees.

The D programming language uses garbage collection for memory management. The practical advantages of using automated memory management are outlined in section 1.2. Like most practical advantages, they are also reflected as economic advantages – less development and testing time allow faster product development and less work expenses.

As this work demonstrates, even automatic memory management is not an universal solution. Even though it is a substantial improvement over manual memory management, it is still possible for problems to occur during application development.

The project described in this report is a study on memory management in the current implementation of the D compiler and the runtime library, as well as a framework for debugging memory problems. Following D's philosophy, this project is *free open-source software*, and as such involved parties have not received material recompension for any work involved.

As outlined in chapter 3, the studied aspects of memory management are performance and two classes of programming errors: memory leaks and memory corruption.

Performance plays an important role in software development. The performance of a certain technological platform greatly affects correct business decisions on the chosen platform for specific projects. It is common to consider that a software development platform's performance is inversely proportional to the typical development time or effort for a specific project. The D programming language tries to lower this factor considerably, by concentrating in performance while providing the productivity of higher-level programming languages.

The performance research in this report describes a performance improvement in certain use cases, which are typical for some types of applications. The significant performance improvement reinforces D's position as a modern, high-performance programming language.

It is no secret that debugging is an important part of the software development cycle. No significantly-sized program is flawless, and software bugs are inevitable. Because of this, software developers need to have access to a set of development and debugging tools which should allow them to quickly and efficiently detect, diagnose and fix programming errors.

The second and third sections of chapter 3 describe two common memory-related software problems: memory leaks and memory corruption. Without the availability of specialized tools, these types of bugs are very tedious and time-consuming to debug.

The product described in chapter 4, "Diamond", is a memory debugger framework for the D programming language. Its goal is to allow programmers to diagnose several types of memory-related software errors in programs written in D quickly and efficiently.

### 5.2 SWOT Analysis

Table 1 SWOT Analysis of the D garbage collector vs. manual memory management

| | |
|---|---|
| **Strengths:**<br>a) immunity against several bugs common with manual memory management:<br>1) conventional memory leaks<br>2) dangling pointer bugs<br>3) double free bugs<br>b) potentially improved performance | **Weaknesses:**<br>a) GC runs are unpredictable (may cause the application to pause sporadically)<br>b) GC run time is unbounded (thus GC cannot be used in realtime applications)<br>c) does not guarantee deallocation when objects go out of scope<br>d) requires distribution as part of the application |
| **Opportunities:**<br>a) Improving the performance of currently available technologies<br>b) Developing new technology in a developing area of computer science | **Threats:**<br>a) Developing technology without existing extensive studies<br>b) May be obsoleted by other techniques |

Table 2 SWOT Analysis of the D programming language

| | |
|---|---|
| **Strengths:**<br>a) high-performance systems programming language<br>b) high-level syntax, allowing rapid prototyping and implementation<br>c) features which reduce the risk of programming errors | **Weaknesses:**<br>a) a comparatively new language without as many reference resources<br>b) lack of corporate backing<br>c) lack of a mature development environment and libraries |
| **Opportunities:**<br>a) One of the most advanced high-level compiled languages<br>b) Can compete with other popular programming languages, such as C++ | **Threats:**<br>a) Developing language with little outside coverage (publications, press)<br>b) Oppressing competition from the much more popular programming languages and existing support for them |

### 5.3 Diamond advantages

Since Diamond is the first memory debugger written specifically for the D programming language, it has no competition and therefore cannot be compared to existing solutions. The following are Diamond's highlights when compared to some solutions for other programming languages:

a) Memory corruption protection – Diamond's memory corruption detection features are specific in that they are cross-platform and do not require interaction with the operating system or direct access with the CPU or other hardware. Diamond uses a simplified approach of "memory stomping" and tracking freed objects, which is both simple and efficient at detecting two common classes of memory management problems.

b) Offline debugging – the Diamond runtime module logs all memory events to a binary log file, allowing inspection after the program has finished executing. This allows the developers to study the log file as many times necessary, without having to re-execute the debugged program and reproduce the problem.

c) Chronologic logging – the log file is designed in such a way as that it were possible to inspect the memory state of a program during any point of the program's execution. Thus, it is possible to determine with a certain precision when any variable in the program's memory was changed.

d) Programmable interface – the Diamond runtime module exposes several functions accessible from the debugged program. This allows programmers to interact with the Diamond runtime module, to specify when to log certain types of events, or to log custom debugging messages which are also saved to the memory log file.

e) Usability – the Diamond analyzer is written for usability in mind, and contains several features aimed at maximizing debugging efficiency (for example, the "top allocators" command).

### 5.4 Time management of the project

The following table shows the time-management of the project. All actions are listed sequentially and have been performed as described below.

Table 3 – Time management pacification of the project

| | Activity name | Duration (days) |
|---|---|---|
| 1 | Project task elaboration | 1 |
| 2 | Definition of task objectives and requirements | 1 |
| 3 | Study of the D programming language | 10 |
| 4 | Study of garbage collection technologies | 5 |
| 5 | Study of tho D garbage collector | 10 |
| 6 | Project plan elaboration | 2 |
| 7 | Design of the future system architecture | 2 |
| 8 | Defining the implementation tasks | 6 |
| 9 | Research and development of the performance improvement | 1 |
| 10 | Design of the Diamond memory log format | 3 |
| 11 | Development of the Diamond module | 5 |
| 12 | Development of the Diamond log analyzer | 5 |
| 13 | Debugging | 2 |
| 14 | Practical application in several projects | 5 |
| 15 | Documentation | 2 |
| 16 | Final testing | 1 |
| 17 | Protection of the labour part elaboration | 3 |
| 18 | Economical part elaboration | 5 |
| 19 | Graphic drafting of the thesis | 3 |
| 20 | Preparation and presentation | 2 |
| | Exception time | 3 |
| | **Total time (days)** | **77** |

Project elaboration duration is estimated to be 74 days. Together with 3 exception days, the total time is 77 working days. It should be taken into consideration the following consultations with the assigned consultants:

Table 4 – Planned consultation duration

| **Nr.** | **Consultant persons** | **Time spent, days** |
|---|---|---|
| 1 | Diploma instructor | 3 |
| 2 | Economical Part Consultant | 3 |
| 3 | Protection of labor consultant | 3 |
| 4 | Standardization consultant | 2 |

### 5.5 Project cost estimation

Calculation of the expenditures is an important aspect of project development, as it decides the profitability of the project. The expenditures for the project realization are classified as follows:

1. stuff expenditures (consumables, raw);
2. wage expenditure (salaries and social contributions of the project developers);
3. other expenditures (amortization of the fixed assets utilized during the project, amortization of non-material assets, fixed assets rent: space, equipment, services: electricity, heat, water, etc.; wages and social contributions of the diploma instructor and the assistants, communications: Internet, phone, fax, GSM, transport expenditures)

#### 5.5.1 Material expenditures (consumables, raw)

The calculations are presented in the table below:

Table 5 Material expenditures

| Nr | Object | Cost p/u | Quantity | Sum, lei | | |
|---|---|---|---|---|---|---|
| | | | | Own source | Extern resource | Total |
| 1 | Computer (Intel Core i7, 4x750 GB RAID, 6 GB RAM) | 12000.0 | 1 | 12000.0 | | 12000.0 |
| 2 | CD-R (700 MB, 52x) | 3.00 | 5 | | 15.0 | 15.0 |
| 3 | CD-RW (700 MB, 10x) | 10.0 | 2 | | 20.0 | 20.0 |
| 4 | Marker (centropen) | 7.0 | 1 | | 7.0 | 7.0 |
| 5 | Pen | 3.00 | 2 | | 6.0 | 6.0 |
| 6 | Notebook, 48 pages | 6.0 | 2 | | 12.0 | 12.0 |
| 7 | Paper(A4 ) 210x297 | 0.10 | 200 | | 20.0 | 20.0 |
| 8 | Microsoft Windows XP Professional W/SP 2 | 1755.6 | 1 | 1755.6 | | 1755.6 |
| 9 | Microsoft Office 2007 | 2860 | 1 | 2860.0 | | 2860.0 |
| 10 | Books | 250.0 | 1 | 250.0 | | 250.0 |
| | **Total** | | | **16865.6** | **80.0** | **16852.1** |

Most of the material expenses were for the high-performance personal computer used to develop the project. Since D is a free language, no expenses have been made in D-related software. A D book has been purchased (“Learn to Tango with D”) for reference.

Software expenses include the Windows operating system and the Microsoft Office environment used for the elaboration of this report. All other software used was free or open-source.

For realization of this project were necessary 200 210x297(A4) papers, 7 CDs, 2 pens, and printer/Xerox/scanner services.

### 5.5.2 Wage expenditures

The work time estimation of the project is **77** working days, with 8 working hours per day. The wage expenditure is composed of the sum of salary of the following persons: the Programmer is paid 6 lei per hour, Project Manager is paid 10 lei per hour and Quality Analyst is paid 8 lei per hour.

Table 6 – Wage expenditures

| Activity carried out | Volume of work,(days) | Wage per unit, (lei per day) | Total wage per function, (lei) |
|---|---|---|---|
| Diploma instructor | 10 | 60 | 600.0 |
| Economic consultant | 3 | 4 | 12.0 |
| Protection consultant | 3 | 4 | 12.0 |
| Standardization consultant | 2 | 4 | 8.0 |
| Project Manager | 77 | 80 | 6160.0 |
| Quality Annalist | 26 | 64 | 1664.0 |
| Programmer | 50 | 48 | 2400.0 |
| Total | | | **10856.0** |
| Medical assurance expenditures (3.5%) | | | 379.96 |
| Social fund expenditures (23%) | | | 2496.88 |
| **Total** | | | **13732.84** |

Sum calculation of the contributions in Social Funds Payments and for Medical assurance:

$SF = F_{ew} * C_{fs} = 10856.0 * 26\% = 2822.56$ (lei)

$MA = F_{ew} * C_{ma\,(\%)} = 10856.0 * 2\% = 217.12$ (lei)

Total sum of the wage expenditures:

$S = F_{ew} + SF + MA = 10856 + 2822.56 + 217.12 =$ **13895.68 (lei)**

### 5.5.3 Indirect expenditures

Table 7 – Indirect expenditures

| Services | Measure, units | Quantity | Cost, per unit | Total cost (lei) |
|---|---|---|---|---|
| Internet | Month | 3 | 80 | 240 |
| Electric energy | kW | 277.2 | 1.1 | 304.92 |
| Telephone | Minutes | 250 | 0.16 | 40 |
| Space rent | Month | 3 | 250 | 750 |
| Transport | Days | 18 | 2 | 36.00 |
| Document Print | Pages | 200 | 0.3 | 60 |
| Document Xerox | Pages | 50 | 0.2 | 10 |
| **Total** | | | | **1440.92** |

For calculating the electric energy was used the next data:

1) Project duration: 77 days;
2) Duration of work per day: 8 hours;
3) Duration of work per day using artificial illumination: 4 hours;
4) Artificial illumination: 3 incandescent bulb 100 W;
5) Power of the equipment: 300 W;
6) Electric energy cost: 0.6 kW/h

The calculation of electricity consumption:

$E_e = 77 * 8 * 300 + 77 * 4 * 300 = 184.800 + 92.400 = 277.200$ (W) = 277,2 (kW)

### 5.5.4 Calculation of the obsolescence of material assets

Table 8 – Obsolescence and amortization of the expenditures

| Nr | Object | Cost p/u, lei | Effective period of function, days | Time of utilization, (days) | Total sum, lei |
|---|---|---|---|---|---|
| 1 | Computer (Intel Celeron, 1GHz,40 GB, 256 MB) | 12000.00 | 1095 | 77 | 843.83 |
| 4 | Microsoft Windows XP Professional W/SP 2 | 1755.6 | 1095 | 77 | 123.45 |
| 6 | Microsoft Office 2007 | 2860 | 1095 | 77 | 201.11 |
| | **Total** | | | | **1168.39** |

**Total Expenditures**

As a result we see that the total expenditures for developing this application and thesis writing are ***16498.42 lei***.

Table 9 Total Expenditures

| **Nr.** | Expenditures | Sum, lei |
|---|---|---|
| 1 | Material expenditures | 80.0 |
| 2 | Wage expenditures | 13895.68 |
| 3 | Indirect expenditures | 1302.32 |
| 5 | Equipment amortization | 1168.39 |
| **Total** | | **16446.39** |

## 5.6 Conclusion

The work described in this report aims at improving an open-source project and describes the elaboration of an open-source project in itself. As such, it has no direct commercial value for the persons involved in the project's development.

The duration of the product elaboration is 77 days, the direct and indirect expenditures determines the total sum of the expenditures that is equal with ***16498.42 lei***. Even though no material gain has been achieved, we can safely say that we have improved the state of the D programming language. As such, future businesses will have a greater incentive to use the D programming language in their projects.

Currently, the state of the D programming language and its community does not allow good conditions for commercial development tools, since most users of the D programming language are open-source developers and are not interested in spending money on commercial applications. However, realizing this product currently has the following long-term economic advantages:

1) it improves the odds of commercial companies using the D programming language as their choice of programming language in commercial products;
2) realization of the project has offered extensive experience in the D programming language and garbage collection, thus opening the possibility of developing high-quality commercial applications in this area.

## 6 Labor and environment protection

(omitted)

# 7 Future Plans

As this paper illustrates, current automated memory management techniques in compiled languages is still far from being perfect, and can be improved in many ways.

## 7.1 Garbage collection

The current D implementation of garbage collection uses a fairly simple and straight-forward approach. As we have seen, it suffers from several disadvantages. One disadvantage not researched in this paper is memory fragmentation.

Memory fragmentation occurs when large objects are allocated intermittently with objects blocks. When several larger objects are freed, it is not possible to allocate an even larger object in the reclaimed space, because there will not be a large enough contiguous area of memory, due the remaining small objects. Memory fragmentation can be prevented by using different pools for objects of different size.

Another idea which would allow implementing a better garbage collector would be being able to track object references. It's not possible to track object references directly without subverting the basic ideals of the D programming language, however it is possible to track writes to large memory areas using features present in modern CPUs and operating systems – page protection. By configuring the operating system's memory manager to trigger an event when the application writes to certain memory pages, it would be possible to track which objects may contain references to changed objects. This could allow implementing new classes of garbage collectors, including "generational" GCs (see section 1.5).

## 7.2 Memory debugging

The Diamond memory debugger is the first implementation of a memory debugger/profiler for the D programming language. As such, it has a comparatively modest feature set.

The next major version of Diamond could move the module inside the runtime library itself, working as a statically compiled patch. Integrating the Diamond module with the garbage collector will allow it more flexibility, allowing the implementation of new memory debugging features, such as memory sentinels.

Another possibility to expand the possibilities of Diamond is integration with *Valgrind*. Valgrind is a programming tool aimed at memory debugging, leak detection and profiling, available for Linux on the Intel x86 platform. Valgrind implements a virtual machine with just-in-time (JIT) capabilities, including dynamic recompilation. When loading a program, Valgrind decompiles the program code to an intermediate representation (an abstract, platform-agnostic code representation), performs necessary changes to the code (such as inserting debug code), then executes it. Although the code runs much slower, this approach allows a great degree of freedom in code modification, including the ability to trace references.

One of Valgrind's features is its ability to track uninitialized memory. Unlike most memory debugging tools, Valgrind does not stop the program whenever it simply accesses any allocated memory that hasn't been initialized. Instead, it remembers which bytes are initialized by the program and which aren't, and propagates this property of memory as the program copies memory around, even in CPU registers. Only when memory is accessed in such a way that its contents could affect execution (for example, it is used in arithmetic or comparison operations), does Valgrind stop the program. This allows for very fine-grained debugging and eliminates many false positives encountered in more straight-forward approaches.

Valgrind allows the programmer to communicate with the Valgrind core to specify manually which areas of memory are initialized or not. Thus, integrating Valgrind with the Diamond module or the D garbage collector has immediate benefits of utilizing Valgrind's powerful abilities in conjunction with D's custom memory management framework.

## Conclusions

As this paper has demonstrated, automatic memory management in compiled languages is still in early stages of development. The only notable implementations are Hans Boehm's garbage collector for C/C++, and the D programming language. Most research in automatic memory management applies to interpreted languages (e.g. Python) or to languages that run inside a virtual machines (Java/.NET).

Nevertheless, an effort has been made to improve the state of one such implementation. Through careful analysis, performance problems have been identified and remedied, thus boosting the performance in those cases significantly.

D's debugging toolchain has been enriched by a versatile memory debugger, Diamond, the development of which is described in this paper. Diamond allows programmers to quickly solve several common problems encountered while developing programs written in the D programming language.

It is certain that the work described will positively impact the state of software development in D. As the quality of software authored in D will improve thankfully to the performance and stability improvements, D will appeal more to new programmers, thus increasing quicker and further adoption.

Finally, it is also clear that the research described in this report is at its early stages, and there is still much to learn and discover about automatic memory management in compiled languages.

## Bibliography


[1] **D 1.0 Change Log** [Online] / auth. Bright Walter // D Programming Language. - http://www.digitalmars.com/d/1.0/changelog.html.

[2] **D 1.0 language specification** [Online] / auth. Bright Walter // D Programming Language. - 2008. - http://www.digitalmars.com/d/1.0/lex.html.

[3] **Dynamic memory allocation and garbage collection** [Journal] / auth. Boehm Hans // Computers in Physics. - May 2005. - Vol. 9. - pp. 297-303.

[4] **Heisenbug** [Online] // Jargon File. - December 29, 2003. - http://www.catb.org/jargon/html/H/heisenbug.html.

[5] **Introduction** [Online] / auth. Bright Walter // D Programming Language. - 2009. - http://www.digitalmars.com/d/.

[6] **Java theory and practice: A brief history of garbage collection** [Online] / auth. Goetz Brian // IBM developerWorks. - October 28, 2003. - http://www.ibm.com/developerworks/java/library/j-jtp10283/.

[7] **Learn to Tango with D** [Book] / auth. Bell Kris [et al.]. - [s.l.] : Apress, 2008.

[8] **Precise detection of memory leaks** [Online] / auth. Maebe Jonas, Michiel Ronsse and De Bosschere Koen // Second International Workshop on Dynamic Analysis. - May 25, 2004. - http://www.cs.virginia.edu/woda2004/papers/maebe.pdf.

[9] **Recursive functions of symbolic expressions and their computation by machine, Part I** [Book Section] / auth. McCarthy John // Communications of the ACM. - New York : ACM, 1960.

[10] **Uniprocessor garbage collection techniques** [Conference] / auth. Wilson P.R. // Proc. Int. Workshop on Memory Management. - St. Malo, France : Springer-Verlag, 1992. - p. 637.

## Annex A – D GC performance improvement patch

This is the patch for the performance improvement described in section 3.1.

```
diff U3 C:/Downloads/dmd.1.028/dmd/src/phobos/internal/gc/gcx.d
         C:/Soft/dmd/src/phobos/internal/gc/gcx.d
--- C:/Downloads/dmd.1.028/dmd/src/phobos/internal/gc/gcx.d    Thu Mar 06 19:31:12
         2008
+++ C:/Soft/dmd/src/phobos/internal/gc/gcx.d    Fri Mar 14 10:04:49 2008
@@ -1750,6 +1750,7 @@
     void **p1 = cast(void **)pbot;
     void **p2 = cast(void **)ptop;
     uint changes = 0;
+    size_t pageCache;

         //printf("marking range: %p -> %p\n", pbot, ptop);
     for (; p1 < p2; p1++)
@@ -1758,8 +1759,10 @@
         byte *p = cast(byte *)(*p1);

         //if (log) debug(PRINTF) printf("\tmark %x\n", p);
-        if (p >= minAddr)
+        if (p >= minAddr && p < maxAddr)
         {
+        if((cast(size_t)p & ~(PAGESIZE-1)) == pageCache)
+            continue;
         pool = findPool(p);
         if (pool)
         {
@@ -1788,6 +1791,8 @@
             // Don't mark bits in B_FREE or B_UNCOMMITTED pages
             continue;
             }
+            if (bin >= B_PAGE)  // cache B_PAGE and B_PAGEPLUS lookups
+            pageCache = cast(size_t)p & ~(PAGESIZE-1);

             //debug(PRINTF) printf("\t\tmark(x%x) = %d\n", biti,
         pool.mark.test(biti));
             if (!pool.mark.test(biti))
```

## Annex B – Diamond module source code listing

This is the source code of the D module which enables memory debugging and memory logging, as described in section 3.2.

```
1.  module diamond;
2.
3.  // options
4.  version = MEMSTOMP;  // stomp on memory when it's freed
5.  version = FREECHECK; // checks manual delete operations
6.
7.  version = MEMLOG;    // log memory operations and content
8.  //version = MEMLOG_VERBOSE; // save memory dumps before and after memory operations
9.  //const MEMLOG_VERBOSE_STEP = 1; // do a full memory dump every ... allocations
10. version = MEMLOG_CRC32; // incremental memory dumps using CRC sums to skip logging memory pages that
    haven't changed between memory dumps
11. const LOGDIR = ``;   // path prefix for memory logs
12.
13. // system configuration
14. version(linux) const _SC_PAGE_SIZE = 30;  // IMPORTANT: may require changing on your platform, look
    it up in your C headers
15.
16. private:
17.
18. version(Tango)
19. {
20.     import tango.core.Memory;
21.     import tango.stdc.stdio;
22.     import tango.stdc.stdlib : stdmalloc = malloc;
23.     version(Windows) import tango.sys.win32.UserGdi : VirtualProtect, PAGE_EXECUTE_WRITECOPY;
24.     else import tango.stdc.posix.sys.mman : mprotect, PROT_READ, PROT_WRITE, PROT_EXEC;
25.     version(MEMLOG) import tango.stdc.time;
26.
27.     // IMPORTANT: add .../tango/lib/gc/basic to the module search path
28.     import gcbits;
29.     import gcx;
30.     import gcstats;
31.     alias gcx.GC GC;
32.
33.     extern (C) void* rt_stackBottom();
34.     alias rt_stackBottom os_query_stackBottom;
35.
36.     extern(C) extern void* D2gc3_gcC3gcx2GC;
37.     alias D2gc3_gcC3gcx2GC gc;
38. }
39. else
40. {
41.     import std.gc;
42.     import std.c.stdio;
43.     import std.c.stdlib : stdmalloc = malloc;
44.     version(Windows) import std.c.windows.windows : VirtualProtect, PAGE_EXECUTE_WRITECOPY;
45.     else import std.c.linux.linux : mprotect, PROT_READ, PROT_WRITE, PROT_EXEC;
46.     version(MEMLOG) import std.c.time;
47.
48.     // IMPORTANT: if the imports below don't work, remove "internal.gc." and add
    ".../dmd/src/phobos/internal/gc" to the module search path
49.     version (Win32) import internal.gc.win32;
50.     version (linux) import internal.gc.gclinux;
51.     import internal.gc.gcbits;
52.     import internal.gc.gcx;
53.     import gcstats;
54.     alias getGCHandle gc;
55. }
56.
57. // configuration ends here
58.
59. // ****************************************************************************
60.
61. struct Array // D underlying array type
62. {
63.     size_t length;
64.     byte *data;
65. }
66.
67. void** ebp()
```

```
68. {
69.     asm
70.     {
71.         naked;
72.         mov EAX, EBP;
73.         ret;
74.     }
75. }
76.
77. public void printStackTrace()
78. {
79.     auto bottom = os_query_stackBottom();
80.     for (void** p=ebp();p;p=cast(void**)*p)
81.     {
82.         printf("%08X\n", *(p+1));
83.         if (*p <= p || *p > bottom)
84.             break;
85.     }
86. }
87.
88. version(MEMLOG)
89. {
90.     FILE* log;
91.
92.     void logDword(uint  i) { fwrite(&i, 4, 1, log); }
93.     void logDword(void* i) { fwrite(&i, 4, 1, log); }
94.     void logData(void[] d) { fwrite(d.ptr, d.length, 1, log); }
95.     void logBits(ref GCBits bits) { logDword(bits.nwords); if (bits.nbits)
    logData(bits.data[1..1+bits.nwords]); }
96.
97.     void logStackTrace()
98.     {
99.         auto bottom = os_query_stackBottom();
100.            for (void** p=ebp();p;p=cast(void**)*p)
101.            {
102.                if (*(p+1))
103.                    logDword(*(p+1));
104.                if (*p <= p || *p > bottom)
105.                    break;
106.            }
107.            logDword(null);
108.        }
109.
110.        enum : int
111.        {
112.            PACKET_MALLOC,
113.            PACKET_CALLOC,
114.            PACKET_REALLOC,
115.            PACKET_EXTEND,
116.            PACKET_FREE,
117.            PACKET_MEMORY_DUMP,
118.            PACKET_MEMORY_MAP,
119.            PACKET_TEXT,
120.            PACKET_NEWCLASS, // metainfo
121.        }
122.
123.        Object logsync;
124.    }
125.
126.    // ****************************************************************************
127.
128.    version(Windows)
129.    {
130.        bool makeWritable(void* address, size_t size)
131.        {
132.            uint old;
133.            return VirtualProtect(address, size, PAGE_EXECUTE_WRITECOPY, &old) != 0;
134.        }
135.    }
136.    else
137.    {
138.        extern (C) int sysconf(int);
139.        bool makeWritable(void* address, size_t size)
140.        {
141.            uint pageSize = sysconf(_SC_PAGE_SIZE);
142.            address = cast(void*)((cast(uint)address) & ~(pageSize-1));
143.            int pageCount = (cast(size_t)address/pageSize == (cast(size_t)address+size)/pageSize) ? 1
    : 2;
144.            return mprotect(address, pageSize * pageCount, PROT_READ | PROT_WRITE | PROT_EXEC) == 0;
145.        }
```

```
146.    }
147.
148.    static uint calcDist(void* from, void* to) { return cast(ubyte*)to - cast(ubyte*)from; }
149.
150.    template Hook(TargetType, HandlerType)
151.    {
152.        static ubyte[] target;
153.        static ubyte[5] oldcode, newcode;
154.        static void initialize(TargetType addr, HandlerType fn)
155.        {
156.            target = cast(ubyte[])(cast(void*)addr)[0..5];
157.            oldcode[] = target;
158.            newcode[0] = 0xE9; // long jump
159.            *cast(uint*)&newcode[1] = calcDist(target.ptr+5, fn);
160.            auto b = makeWritable(target.ptr, target.length);
161.            assert(b);
162.            hook();
163.        }
164.
165.        static void hook() { target[] = newcode; }
166.        static void unhook() { target[] = oldcode; }
167.    }
168.
169.    /// Hook a function by overwriting the first bytes with a jump to your handler. Calls the original
    by temporarily restoring the hook (caller needs to do that manually due to the way arguments are
    passed on).
170.    /// WARNING: this may only work with the calling conventions specified in the D documentation (
    http://www.digitalmars.com/d/1.0/abi.html ), thus may not work with GDC
171.    struct FunctionHook(int uniqueID, ReturnType, Args ...)
172.    {
173.        mixin Hook!(ReturnType function(Args), ReturnType function(Args));
174.    }
175.
176.    /// The last argument of the handler is the context.
177.    struct MethodHook(int uniqueID, ReturnType, ContextType, Args ...)
178.    {
179.        mixin Hook!(ReturnType function(Args), ReturnType function(Args, ContextType));
180.    }
181.
182.    /// Hook for extern(C) functions.
183.    struct CFunctionHook(int uniqueID, ReturnType, Args ...)
184.    {
185.        extern(C) alias ReturnType function(Args) FunctionType;
186.        mixin Hook!(FunctionType, FunctionType);
187.    }
188.
189.    MethodHook!(1, size_t, Gcx*, void*) fullcollectHook;
190.    version(MEMSTOMP)
191.    {
192.        CFunctionHook!(2, byte[], TypeInfo, size_t, Array*) arraysetlengthTHook;
193.        CFunctionHook!(3, byte[], TypeInfo, size_t, Array*) arraysetlengthiTHook;
194.    }
195.    version(MEMLOG)
196.    {
197.        CFunctionHook!(1, Object, ClassInfo) newclassHook;
198.    }
199.
200.    // ****************************************************************************
201.
202.    void enforce(bool condition, char[] message)
203.    {
204.        if (!condition)
205.        {
206.            //printStackTrace();
207.            throw new Exception(message);
208.        }
209.    }
210.
211.    final class DiamondGC : GC
212.    {
213.        // note: we can't add fields here because we are overwriting the original class's virtual call
    table
214.
215.        final void mallocHandler(size_t size, void* p)
216.        {
217.            //printf("Allocated %d bytes at %08X\n", size, p); printStackTrace();
218.            version(MEMLOG) synchronized(logsync)
219.                if (p)
220.                {
221.                    logDword(PACKET_MALLOC);
```

```
                    logDword(time(null));
                    logStackTrace();
                    logDword(p);
                    logDword(size);
                }
            version(MEMLOG_VERBOSE) verboseLog();
        }

        final void callocHandler(size_t size, void* p)
        {
            //printf("Allocated %d initialized bytes at %08X\n", size, p); printStackTrace();
            version(MEMLOG) synchronized(logsync)
                if (p)
                {
                    logDword(PACKET_CALLOC);
                    logDword(time(null));
                    logStackTrace();
                    logDword(p);
                    logDword(size);
                }
            version(MEMLOG_VERBOSE) verboseLog();
        }

        final void reallocHandler(size_t size, void* p1, void* p2)
        {
            //printf("Reallocated %d bytes from %08X to %08X\n", size, p1, p2); printStackTrace();
            version(MEMLOG) synchronized(logsync)
                if (p2)
                {
                    logDword(PACKET_REALLOC);
                    logDword(time(null));
                    logStackTrace();
                    logDword(p1);
                    logDword(p2);
                    logDword(size);
                }
            version(MEMLOG_VERBOSE) verboseLog();
        }

        override size_t extend(void* p, size_t minsize, size_t maxsize)
        {
            auto result = super.extend(p, minsize, maxsize);
            version(MEMLOG) synchronized(logsync)
                if (result)
                {
                    logDword(PACKET_EXTEND);
                    logDword(time(null));
                    logStackTrace();
                    logDword(p);
                    logDword(result);
                }
            version(MEMLOG_VERBOSE) verboseLog();
            return result;
        }

        override void free(void *p)
        {
            version(FREECHECK)
            {
                Pool* pool = gcx.findPool(p);
                enforce(pool !is null, "Freed item is not in a pool");

                uint pagenum = (p - pool.baseAddr) / PAGESIZE;
                Bins bin = cast(Bins)pool.pagetable[pagenum];
                enforce(bin <= B_PAGE, "Freed item is not in an allocated page");

                size_t size = binsize[bin];
                enforce((cast(size_t)p & (size - 1)) == 0, "Freed item is not aligned to bin
    boundary");

                if (bin < B_PAGE)  // Check that p is not on a free list
                    for (List *list = gcx.bucket[bin]; list; list = list.next)
                        enforce(cast(void *)list != p, "Freed item is on a free list");
            }
            version(MEMLOG) synchronized(logsync)
            {
                logDword(PACKET_FREE);
                logDword(time(null));
                logStackTrace();
                logDword(p);
```

```
301.            }
302.            version(MEMLOG_VERBOSE) verboseLog();
303.            version(MEMSTOMP)
304.            {
305.                auto c = capacity(p);
306.                super.free(p);
307.                if (c>4)
308.                    (cast(ubyte*)p)[4..c] = 0xBD;
309.            }
310.            else
311.                super.free(p);
312.            version(MEMLOG_VERBOSE) verboseLog();
313.        }
314.
315.        version(Tango)
316.        {
317.            override void *malloc(size_t size, uint bits) { version(MEMLOG_VERBOSE) verboseLog(); auto result = super.malloc(size, bits); mallocHandler(size, result); return result; }
318.            override void *calloc(size_t size, uint bits) { version(MEMLOG_VERBOSE) verboseLog(); auto result = super.calloc(size, bits); callocHandler(size, result); return result; }
319.            override void *realloc(void *p, size_t size, uint bits) { version(MEMLOG_VERBOSE) verboseLog(); auto result = super.realloc(p, size, bits); reallocHandler(size, p, result); return result; }
320.            alias sizeOf capacity;
321.        }
322.        else
323.        {
324.            override void *malloc(size_t size) { version(MEMLOG_VERBOSE) verboseLog(); auto result = super.malloc(size); mallocHandler(size, result); return result; }
325.            override void *calloc(size_t size, size_t n) { version(MEMLOG_VERBOSE) verboseLog(); auto result = super.calloc(size, n); callocHandler(size*n, result); return result; }
326.            override void *realloc(void *p, size_t size) { version(MEMLOG_VERBOSE) verboseLog(); auto result = super.realloc(p, size); reallocHandler(size, p, result); return result; }
327.        }
328.    }
329.
330.    version(MEMLOG)
331.    {
332.        const uint FORMAT_VERSION = 2; // format of the log file
333.
334.        version(MEMLOG_CRC32)
335.        {
336.            const MAX_POOLS = 1024;
337.            uint*[MAX_POOLS] poolCRCs;
338.
339.            uint[256] crc32_table =
    [0x00000000,0x77073096,0xee0e612c,0x990951ba,0x076dc419,0x706af48f,0xe963a535,0x9e6495a3,0x0edb8832,0x79dcb8a4,0xe0d5e91e,0x97d2d988,0x09b64c2b,0x7eb17cbd,0xe7b82d07,0x90bf1d91,0x1db71064,0x6ab020f2,0xf3b97148,0x84be41de,0x1adad47d,0x6ddde4eb,0xf4d4b551,0x83d385c7,0x136c9856,0x646ba8c0,0xfd62f97a,0x8a65c9ec,0x14015c4f,0x63066cd9,0xfa0f3d63,0x8d080df5,0x3b6e20c8,0x4c69105e,0xd56041e4,0xa2677172,0x3c03e4d1,0x4b04d447,0xd20d85fd,0xa50ab56b,0x35b5a8fa,0x42b2986c,0xdbbbc9d6,0xacbcf940,0x32d86ce3,0x45df5c75,0xdcd60dcf,0xabd13d59,0x26d930ac,0x51de003a,0xc8d75180,0xbfd06116,0x21b4f4b5,0x56b3c423,0xcfba9599,0xb8bda50f,0x2802b89e,0x5f058808,0xc60cd9b2,0xb10be924,0x2f6f7c87,0x58684c11,0xc1611dab,0xb6662d3d,0x76dc4190,0x01db7106,0x98d220bc,0xefd5102a,0x71b18589,0x06b6b51f,0x9fbfe4a5,0xe8b8d433,0x7807c9a2,0x0f00f934,0x9609a88e,0xe10e9818,0x7f6a0dbb,0x086d3d2d,0x91646c97,0xe6635c01,0x6b6b51f4,0x1c6c6162,0x856530d8,0xf262004e,0x6c0695ed,0x1b01a57b,0x8208f4c1,0xf50fc457,0x65b0d9c6,0x12b7e950,0x8bbeb8ea,0xfcb9887c,0x62dd1ddf,0x15da2d49,0x8cd37cf3,0xfbd44c65,0x4db26158,0x3ab551ce,0xa3bc0074,0xd4bb30e2,0x4adfa541,0x3dd895d7,0xa4d1c46d,0xd3d6f4fb,0x4369e96a,0x346ed9fc,0xad678846,0xda60b8d0,0x44042d73,0x33031de5,0xaa0a4c5f,0xdd0d7cc9,0x5005713c,0x270241aa,0xbe0b1010,0xc90c2086,0x5768b525,0x206f85b3,0xb966d409,0xce61e49f,0x5edef90e,0x29d9c998,0xb0d09822,0xc7d7a8b4,0x59b33d17,0x2eb40d81,0xb7bd5c3b,0xc0ba6cad,0xedb88320,0x9abfb3b6,0x03b6e20c,0x74b1d29a,0xead54739,0x9dd277af,0x04db2615,0x73dc1683,0xe3630b12,0x94643b84,0x0d6d6a3e,0x7a6a5aa8,0xe40ecf0b,0x9309ff9d,0x0a00ae27,0x7d079eb1,0xf00f9344,0x8708a3d2,0x1e01f268,0x6906c2fe,0xf762575d,0x806567cb,0x196c3671,0x6e6b06e7,0xfed41b76,0x89d32be0,0x10da7a5a,0x67dd4acc,0xf9b9df6f,0x8ebeeff9,0x17b7be43,0x60b08ed5,0xd6d6a3e8,0xa1d1937e,0x38d8c2c4,0x4fdff252,0xd1bb67f1,0xa6bc5767,0x3fb506dd,0x48b2364b,0xd80d2bda,0xaf0a1b4c,0x36034af6,0x41047a60,0xdf60efc3,0xa867df55,0x316e8eef,0x4669be79,0xcb61b38c,0xbc66831a,0x256fd2a0,0x5268e236,0xcc0c7795,0xbb0b4703,0x220216b9,0x5505262f,0xc5ba3bbe,0xb2bd0b28,0x2bb45a92,0x5cb36a04,0xc2d7ffa7,0xb5d0cf31,0x2cd99e8b,0x5bdeae1d,0x9b64c2b0,0xec63f226,0x756aa39c,0x026d930a,0x9c0906a9,0xeb0e363f,0x72076785,0x05005713,0x95bf4a82,0xe2b87a14,0x7bb12bae,0x0cb61b38,0x92d28e9b,0xe5d5be0d,0x7cdcefb7,0x0bdbdf21,0x86d3d2d4,0xf1d4e242,0x68ddb3f8,0x1fda836e,0x81be16cd,0xf6b9265b,0x6fb077e1,0x18b74777,0x88085ae6,0xff0f6a70,0x66063bca,0x11010b5c,0x8f659eff,0xf862ae69,0x616bffd3,0x166ccf45,0xa00ae278,0xd70dd2ee,0x4e048354,0x3903b3c2,0xa7672661,0xd06016f7,0x4969474d,0x3e6e77db,0xaed16a4a,0xd9d65adc,0x40df0b66,0x37d83bf0,0xa9bcae53,0xdebb9ec5,0x47b2cf7f,0x30b5ffe9,0xbdbdf21c,0xcabac28a,0x53b39330,0x24b4a3a6,0xbad03605,0xcdd70693,0x54de5729,0x23d967bf,0xb3667a2e,0xc4614ab8,0x5d681b02,0x2a6f2b94,0xb40bbe37,0xc30c8ea1,0x5a05df1b,0x2d02ef8d];
340.            uint fastCRC(void[] data) // we can't use the standard Phobos crc32 function because we can't rely on inlining being available (because it's natural to compile debuggees without optimizations), and calling a function for every byte would be too slow
341.            {
342.                uint crc = cast(uint)-1;
343.                foreach (ubyte val;cast(ubyte[])data)
```

```
                    crc = crc32_table[cast(ubyte) crc ^ val] ^ (crc >> 8);
                return crc;
            }
        }

        extern(C) public void logMemoryDump(bool dataDump, Gcx* gcx = null)
        {
            synchronized(logsync)
            {
                //dataDump ? printf("Dumping memory contents...\n") : printf("Dumping memory map...\n");
                if (gcx is null) gcx = (cast(GC)gc).gcx;
                logDword(dataDump ? PACKET_MEMORY_DUMP : PACKET_MEMORY_MAP);
                logDword(time(null));
                logStackTrace();
                logDword(gcx.npools);
                for (int pn=0;pn<gcx.npools;pn++)
                {
                    auto p = gcx.pooltable[pn];
                    logDword(p.baseAddr);
                    logDword(p.npages);
                    logDword(p.ncommitted);
                    logData(p.pagetable[0..p.npages]);
                    logBits(p.freebits);
                    logBits(p.finals);
                    logBits(p.noscan);
                    if (dataDump)
                    {
                        version(MEMLOG_CRC32)
                        {
                            assert(pn < MAX_POOLS);
                            if (poolCRCs[pn] is null)
                            {
                                poolCRCs[pn] = cast(uint*)stdmalloc(4*p.npages);
                                poolCRCs[pn][0..p.npages] = 0;
                            }
                        }
                        for (int pg=0;pg<p.ncommitted;pg++)
                        {
                            bool doSave = true;
                            auto page = p.baseAddr[pg*PAGESIZE..(pg+1)*PAGESIZE];
                            version(MEMLOG_CRC32)
                            {
                                uint newCRC = fastCRC(page);
                                if (newCRC==poolCRCs[pn][pg] && newCRC!=0)
                                    doSave = false;
                                else
                                    poolCRCs[pn][pg] = newCRC;
                            }
                            logDword(doSave?1:0);
                            if (doSave)
                                logData(page);
                        }
                    }
                }
                if (dataDump)
                    logData(gcx.bucket);
                fflush(log);
                //printf("Done\n");
            }
        }

        version(MEMLOG_VERBOSE)
            void verboseLog()
            {
                static int n = 0;
                if (n++ % MEMLOG_VERBOSE_STEP == 0)
                    logMemoryDump(true);
            }

        extern(C) public void logText(char[] text)
        {
            synchronized(logsync)
            {
                logDword(PACKET_TEXT);
                logDword(time(null));
                logStackTrace();
                logDword(text.length);
                logData(text);
            }
```

```
423.        }
424.
425.        extern(C) public void logNumber(uint n)
426.        {
427.            char[24] buf;
428.            sprintf(buf.ptr, "%08X (%d)", n, n);
429.            for (int i=12;i<buf.length;i++)
430.                if (!buf[i])
431.                    return logText(buf[0..i]);
432.        }
433.    }
434.
435.    size_t fullcollectHandler(void* stackTop, Gcx* gcx)
436.    {
437.        //printf("minaddr=%08X maxaddr=%08X\n", gcx.minAddr, gcx.maxAddr);
438.        //printf("Beginning garbage collection\n");
439.        version(MEMLOG) logMemoryDump(true, gcx);
440.        fullcollectHook.unhook();
441.        auto result = gcx.fullcollect(stackTop);
442.        fullcollectHook.hook();
443.        version(MEMLOG) logMemoryDump(false, gcx);
444.        //printf("Garbage collection done, %d pages freed\n", result);
445.        return result;
446.    }
447.
448.    version(MEMSTOMP)
449.    {
450.        // stomp on shrunk arrays
451.
452.        extern(C) extern byte[] _d_arraysetlengthT(TypeInfo ti, size_t newlength, Array *p);
453.        extern(C) extern byte[] _d_arraysetlengthiT(TypeInfo ti, size_t newlength, Array *p);
454.
455.        extern(C) byte[] arraysetlengthTHandler(TypeInfo ti, size_t newlength, Array *p)
456.        {
457.            Array old = *p;
458.            arraysetlengthTHook.unhook();
459.            auto result = _d_arraysetlengthT(ti, newlength, p);
460.            arraysetlengthTHook.hook();
461.            //printf("_d_arraysetlengthT: %d => %d\n", oldlength, p.length);
462.            size_t sizeelem = ti.next.tsize();
463.            if (old.data == p.data && p.length < old.length)
464.                (cast(ubyte*)p.data)[p.length*sizeelem .. old.length*sizeelem] = 0xBD;
465.            return result;
466.        }
467.
468.        extern(C) byte[] arraysetlengthiTHandler(TypeInfo ti, size_t newlength, Array *p)
469.        {
470.            Array old = *p;
471.            arraysetlengthiTHook.unhook();
472.            auto result = _d_arraysetlengthiT(ti, newlength, p);
473.            arraysetlengthiTHook.hook();
474.            //printf("_d_arraysetlengthiT: %d => %d\n", oldlength, p.length);
475.            size_t sizeelem = ti.next.tsize();
476.            if (old.data == p.data && p.length < old.length)
477.                (cast(ubyte*)p.data)[p.length*sizeelem .. old.length*sizeelem] = 0xBD;
478.            return result;
479.        }
480.    }
481.
482.    version(MEMLOG)
483.    {
484.        extern(C) extern Object _d_newclass(ClassInfo ci);
485.
486.        extern(C) Object newclassHandler(ClassInfo ci)
487.        {
488.            if ((ci.flags & 1)==0)
489.                synchronized(logsync)
490.                {
491.                    logDword(PACKET_NEWCLASS);
492.                    logDword(ci.name.length);
493.                    logData(ci.name);
494.                }
495.            newclassHook.unhook();
496.            auto result = _d_newclass(ci);
497.            newclassHook.hook();
498.            return result;
499.        }
500.
501.    }
502.
```

```
503.    // ****************************************************************************
504.
505.    static this()
506.    {
507.        version(MEMLOG) logsync = new Object;
508.        // replace the garbage collector Vtable
509.        *cast(void**)gc = DiamondGC.classinfo.vtbl.ptr;
510.
511.        fullcollectHook.initialize(&Gcx.fullcollect, &fullcollectHandler);
512.        version(MEMSTOMP)
513.        {
514.            arraysetlengthTHook.initialize(&_d_arraysetlengthT, &arraysetlengthTHandler);
515.            arraysetlengthiTHook.initialize(&_d_arraysetlengthiT, &arraysetlengthiTHandler);
516.        }
517.        version(MEMLOG)
518.        {
519.            newclassHook.initialize(&_d_newclass, &newclassHandler);
520.            time_t t = time(null);
521.            tm *tm = localtime(&t);
522.            char[256] name;
523.            sprintf(name.ptr, "%sdiamond_%d-%02d-%02d_%02d.%02d.%02d.mem",
    LOGDIR.length?LOGDIR.ptr:"", 1900+tm.tm_year, tm.tm_mon, tm.tm_mday, tm.tm_hour, tm.tm_min,
    tm.tm_sec);
524.            log = fopen(name.ptr, "wb");
525.            logDword(FORMAT_VERSION);
526.        }
527.    }
528.
529.    static ~this()
530.    {
531.        version(MEMLOG)
532.        {
533.            //printf("Closing memory log...\n");
534.            fclose(log);
535.        }
536.    }
```