Advances in Programming Languages: Memory management

Stephen Gilmore
The University of Edinburgh

January 15, 2007

Memory management

Computer programs need to allocate memory to store data values and data structures. Memory is also used to store the program itself and the run-time system needed to support it.

If a program allocates memory and never frees it, and that program *runs for a sufficiently long time*, eventually it will run out of memory.

Even in the presence of *virtual memory*, memory consumption is still a major issue because it is considerably less efficient to access virtual memory than to access physical memory.

Manual and automatic memory management

Programming languages can be categorised as those which provide automatic memory management and those which ask the programmer to allocate and free memory manually.

Requiring the programmer to do the work manually leads to a simpler compiler and runtime. The C language requires the programmer to implement memory management each time, for each application program. Modern programming languages such as Java, C#, Caml, Cyclone and Ruby provide automatic memory management with *garbage collection*.

Manual memory management

In C, where there is no garbage collector, the programmer must allocate and free memory explicity. The key functions are `malloc` and `free`. The `malloc` function takes as a parameter the size in bytes of the memory area to be allocated. The size of a type can be obtained using `sizeof`.

The resulting area of memory does not represent a value of the correct type so it then needs to be *cast* to the correct type.

```
  p = (Type_t*) malloc(sizeof(Type_t));
```

A significant problem with manual memory management is that it is possible to attempt to use a pointer after it has been freed. This is known as the *dangling pointer problem.* Dangling pointer errors can arise whenever there is an error in the control flow logic of a program. This can lead to allocation, use and deallocation happening in the wrong order in some circumstances.

Use before allocation may be a fatal run-time error. Use after deallocation is not always fatal. *Neither of these is a good thing.*

```
/* File: programs/c/DanglingPointers.c */
#include <stdio.h>
#include <stdlib.h>
```

```
typedef struct {       /*  define a structure */
  int x;               /* ... with an x field */
  int y;               /*   ... and a y field */
} Coordinate_t;    /*  ... called Coordinate_t */

int main() {
  /* Allocate a pointer to a coordinate */
  Coordinate_t *p;
  p = (Coordinate_t*)malloc(sizeof(Coordinate_t));

  /* Use p */
  p->x = 256; /* Or: (*p).x = 256; */
  p->y = 512; /* Or: (*p).y = 512; */
  printf("p->x is %d\n", p->x); /* "p->x is 256" */
  printf("p->y is %d\n", p->y); /* "p->y is 512" */
  /* Deallocate p */
  free(p);
  /* Erroneous attempt to use p after deallocation */
  printf("p->x is %d\n", p->x); /* "p->x is 0" */
  printf("p->y is %d\n", p->y); /* "p->y is 512" */

  /* Allocate another pointer to a coordinate */
  Coordinate_t *p2;
  p2 = (Coordinate_t*)malloc(sizeof(Coordinate_t));

  /* Erroneous attempt to use p2 before initialisation */
  printf("p2->x is %d\n", p2->x); /* "p2->x is 0" */
  printf("p2->y is %d\n", p2->y); /* "p2->y is 512" */

  /* Update p2 */
  p2->x = 1024;

  /* Erroneous attempt to use p after deallocation */
  printf("p->x is %d\n", p->x); /* "p->x is 1024" */
  printf("p->y is %d\n", p->y); /* "p->y is 512" */

  exit(0);
}
```

The result of this program is compiler-dependent. Some C compilers will print different results for the values pointed to after `free` is called.

Another potential problem of manual memory management is *not* remembering to free allocated memory when it should be freed. The reference to an allocated area of memory can be lost when a variable in a block-structured language goes out of scope. This problem is perhaps more subtle than the dangling pointer problem because it may only become manifest for long-running applications. When memory is lost and cannot be reclaimed we term this a *space leak*. Space cannot be lost forever without reaching the limit on the available memory. A long-running program with a space leak will eventually crash.

```
/* File: programs/c/Memory.c */
#include <stdio.h>
#include <stdlib.h>

typedef struct {       /*  define a structure */
  float values[1000]; /*  ... of 1000 floats */
} Vector_t;           /* ... called Vector_t */

int main() {
  Vector_t *v;
  /* allocate memory unceasingly */
  for (;;) v = (Vector_t*)malloc(sizeof(Vector_t));
  exit(0);
}
```

Never run this program. On a typical Linux platform, this program will allocate memory very rapidly, filling up the available real memory. Then the *Kernel Swap Daemon* (`kswapd`) will be invoked to swap pages of memory out to the swap file. Fairly soon, the swap file fills up and the program may be killed by the operating system (thereby freeing up all of the memory which it claimed).

The effect of attempting to allocate memory when there is no more left to be allocated depends ultimately on the definition of the `malloc` function. The `malloc` function is defined to return a null pointer when it cannot allocate the required memory. Potentially any call to `malloc` in a C program must be prepared to deal with a null pointer being returned as a result.

Memory problems and solutions

The C technology chose to keep the language compiler and run-time as lean as possible, designing for much less powerful computing technology than we typically have at our disposal today. One example of this was that static analysis and program inspection routines were moved out of the compilers into separate tools such as `lint`. This analysis was so useful that it is now typically re-integrated into C compilers (`gcc -Wall` performs lint-like static analysis of C programs). Separate tools such as *Purify* are used to detect memory-related problems in a lint-like fashion.

Perhaps a good way to think about C is that it is a programming language which treats the developer as a grown-up. It is not very well-suited as a programming language for beginners to use. It does not warn about a lot of potential problems at compile time. Then at run-time when problems occur they might either be silently ignored or terminate the application. So programming in C is a bit like breaking the law: you might not get caught. (But if you do it's the death penalty.)

Array out-of-bounds violations in C

The C programming language is not supported by a well-managed run-time such as the virtual machines which Java, O'Caml and Ruby have, or the common language run-time of .NET used by C#. No run-time type-checking is taking place as a C program executes. There is no Security Manager. No-one is tracking array bounds violations. The consequence of this is that C programs may contain hidden errors which generate no compile-time warnings and which do not show up in testing. The bad consequences of this are well-known; code with undiscovered bugs is signed off by the developer and shipped to the customer, only to go wrong later when it is used.

```
/* File: programs/c/ArrayViolation.c */
#include <stdio.h>
#include <malloc.h>

int main() {
  /* An array of four integers */
  int* squares = (int*) malloc (4 * sizeof(int));
  int i;
  for (i = 1 ; i <= 4; i++) /* initialise the array */
    squares[i] = i * i;
  for (i = 1 ; i <= 4; i++) /* print the contents */
    printf("%d\n", squares[i]);
  return(0);
}
```

The error in this program is that arrays in C are indexed from zero, and so the access to `a[4]` is out of bounds (an off-by-one error). C does not warn us about this at compile time (most other languages would not either). The compile command `gcc -Wall -o arrayviolation ArrayViolation.c` produces no warnings. However, neither does it fail at run-time.

```
[scap]stg: ./arrayviolation
1
4
9
16
[scap]stg:
```

In contrast, the corresponding Java program

```
/* File: programs/java/ArrayViolation.java */
public class ArrayViolation {
  public static void main(String[] args) {
    /* An array of four integers */
    int[] squares = new int[4];
    int i;
    for (i = 1 ; i <= 4; i++) /* initialise the array */
      squares[i] = i * i;
    for (i = 1 ; i <= 4; i++) /* print the contents */
      System.out.printf("%d\n", squares[i]);
  }
}
```

fails at run-time when trying to initialise the array.

```
[scap]stg: java ArrayViolation
Exception in thread "main"
   java.lang.ArrayIndexOutOfBoundsException: 4
      at ArrayViolation.main(ArrayViolation.java:8)
```

In general, applications which do a significant amount of array processing may be slower in high-level languages such as Java, C# and O'Caml, because of bounds checking, but it is easier for developers to find bugs in these programs during testing, which may lead to code of higher quality being shipped to the customer.

Modern implementations of `malloc` on the Linux platform allow the user to influence the behaviour of `malloc` by setting the `MALLOC_CHECK_` environment variable to indicate how strictly errors such as the off-by-one error in the program above should be penalised.

Languages built on libraries

Because C is a flexible language, built on the concept of *libraries* of externally accessible functions, one can experiment with alternative memory management routines without needing to modify the compiler.

In Java, where the memory-management routines are baked-in, application developers do not have this flexibility. For example, it is not possible for Java application programmers to re-define the meaning of the “new” operation.

Detecting bounds violations

A way to make good use of this flexibility in C is to replace `malloc` with a more strict implementation. One such is *Electric Fence* which stops the program immediately once the allocated area is overrun. It is not necessary to re-compile an application to use the library, simply add it to the load path.

```
[scap]stg: export LD_PRELOAD=libefence.so.0
[scap]stg: ./arrayviolation

  Electric Fence 2.2.0 Copyright (C) 1987-1999
  Bruce Perens <bruce@perens.com>
Segmentation fault
```

The Electric Fence library traps errors such as the dangling pointer error which we saw previously. Using a pointer after it has been freed is now an immediate segmentation fault.

```
[scap]stg: ./danglingpointers

  Electric Fence 2.2.0 Copyright (C) 1987-1999
  Bruce Perens <bruce@perens.com>
p->x is 256
p->y is 512
Segmentation fault
[scap]stg:
```

Electric Fence uses virtual memory hardware to place an inaccessible memory page after (and optionally before) any memory allocation. Any attempt to even read from these locations generates a segmentation fault.

Similarly memory released by `free()` is made inaccessible (and will not be reallocated) so that any attempt to access memory via a freed pointer is illegal. The pointer is not updated.

Electric Fence makes very wasteful use of memory and is suitable only for debugging, not for production code.

Detecting space leaks

Electric Fence detects errors in the “dangling pointer” class, but not space leaks. As a more typical example of a space leak than the one which we saw previously, the following program leaks a vector of 1000 floating point numbers. This occurs because it fails to free the allocated memory before the relevant function exits and the local variable holding the pointer to the allocated memory is destroyed.

```
/* File: programs/c/SpaceLeak.c */
#include <stdio.h>
#include <stdlib.h>

typedef struct {      /*  define a structure */
  float values[1000]; /*  ... of 1000 floats */
} Vector_t;           /* ... called Vector_t */

float min(Vector_t v) {
  float x = v.values[0];
  int i;
  for (i = 1 ; i < 1000 ; i++)
    if (v.values[i] < x)
      x = v.values[i];
  return x;
}

/* This function has a (rather artificial) space leak
   which occurs only because we allocate the local
   vector v dynamically using a call to malloc */
float biasedRnd() {
  Vector_t *v = (Vector_t*)malloc(sizeof(Vector_t));
  int i;
  for (i = 0 ; i < 1000 ; i++)
    v->values[i] = (float)rand();
  return min(*v);
  /* Space leak: We forgot to free v! */
}

int main() {
  printf("Biased random number (least of 1000): %d\n",
         (int)biasedRnd());
  exit(0);
}
```

The `valgrind` debugger allows us to find memory leaks using the `memcheck` memory checker.

We compile the program as usual with `gcc -Wall -o spaceleak SpaceLeak.c`. We run the program using `valgrind spaceleak`.

When our program finishes we receive a summary of the space leaked.

```
== LEAK SUMMARY:
==    definitely lost: 4000 bytes in 1 blocks.
```

This program can be easily repaired by replacing the unnecessary call to `malloc`.

```
/* File: programs/c/SpaceLeak2.c */
#include <stdio.h>
#include <stdlib.h>

typedef struct {      /*  define a structure */
  float values[1000]; /*  ... of 1000 floats */
} Vector_t;           /* ... called Vector_t */

float min(Vector_t v) {
  float x = v.values[0];
  int i;
  for (i = 1 ; i < 1000 ; i++)
    if (v.values[i] < x)
      x = v.values[i];
  return x;
}

/* This function does not have a space leak because
   we allocate the vector on the stack rather than on
   the heap, so we do not call malloc at all */
float biasedRnd() {
  Vector_t v;
  int i;
  for (i = 0 ; i < 1000 ; i++)
    v.values[i] = (float)rand();
  return min(v);
}

int main() {
  printf("Biased random number (least of 1000): %d\n",
         (int)biasedRnd());
  exit(0);
}
```

For this program `valgrind` reports

```
No malloc'd blocks -- no leaks are possible.
```

One way to find space leaks in C programs would be to use the *Debug Malloc* library (`dmalloc.com`). This also implements bounds-checking by *fence-posting* data structures by placing distinguished markers both before and after the allocated area. Overwriting such a marker is faulted at run-time. Some C compilers provide a limited version of fence-posts (*stack guards*) which delimit data areas in stack frames, thereby preventing overwriting the return address of a function.

Errors such as freeing pointers too soon can be fixed by using the Boehm-Demers-Weiser garbage collector (`www.hpl.hp.com/personal/Hans_Boehm/gc/`).

The Boehm collector maintains a record of pointer data structures so that live data can be recovered (whereas discarded data can be removed in a garbage collection). The effect on run-time is that calls to `malloc` become more expensive, whereas `free` operations are cheaper (essentially a no-op). It is then no longer an error to call `free` on a pointer which has already been freed.

Garbage collection in Cyclone

Cyclone uses the Boehm collector so we can run the previous C program unmodified in Cyclone.

```
/* File: programs/cyclone/Memory.cyc */
#include <stdio.h>
#include <stdlib.h>

typedef struct {      /*  define a structure */
  float values[1000]; /*  ... of 1000 floats */
} Vector_t;           /* ... called Vector_t */

int main() {
  Vector_t *v;
  /* allocate memory unceasingly */
  for (;;) v = (Vector_t*)malloc(sizeof(Vector_t));
  exit(0);
}
```

The run-time behaviour of this program is very different in Cyclone. Allocated memory is reclaimed by periodic garbage collections and the Kernel Swap Daemon is never invoked. In Cyclone this program does not exhaust physical memory and swap memory and does not need to be killed by the operating system.

Allocation and collection in Java

The Java runtime has a garbage collector which separates data (by age) into *generations*. Both *stop-and-copy* and *mark-and-sweep* collectors are implemented in SUN's JVM. The GNU Java compiler, `gcj`, uses the Boehm collector. Helpfully, the SUN Java runtime allows us to inspect the results (and run-time performance cost) of collections. We run the following with `java -verbose:gc`.

```
/* File: programs/java/Memory.java */
class Memory {
   public static void main(String[] args) {
       /* Nested class in main method */
       class Vector {
          float[] values = new float[1000];
       }

       Vector v;
       /* allocate memory unceasingly */
       for (;;) v = new Vector();
       /* System.exit(0); <- unreachable statement */
   }
}
```

Allocation and collection in C#

The C# language depends on a garbage collector to reclaim unused dynamic data structures in memory, as Java does.

The following example was compiled and tested with the Ximian Mono compiler for C#, available for Linux and other Unix platforms (`go-mono.com` and `gotmono.com`). The Mono compiler also depends on the Boehm collector.

```
/* File: programs/cs/Memory.cs */
using System;
class Memory {
  /* No nested classes in C# */
  public class Vector {
    float[] values = new float[1000];
  }
  public static void Main(string[] args) {
    Vector v;
    /* allocate memory unceasingly */
    for (;;) v = new Vector();
    Environment.ExitCode = 0; // generates a warning
  }
}
```

Allocation and collection in Caml

Objective Caml depends on a garbage collector developed by Damien Doligez and Xavier Leroy. Our example program is shorter in Caml because the types of variables are inferred by the compiler.

```
(* File: programs/caml/Memory.ml *)
let rec main() =
  (* allocate memory unceasingly *)
  let v = Array.create 1000 0.0
  in main();;

main(); exit 0;;
```

Allocation and collection in Ruby

The Ruby programming language has an integrated mark-and-sweep garbage collector. Like Caml, Ruby programs are concise.

```
# File: programs/ruby/Memory.rb
# allocate memory unceasingly
while true
  v = Array.new(1000, 0.0)
end
```

Summary

- C, like many other languages, asks programmers to manage memory allocation themselves.
- Manual memory management is an error-prone process. We may call `free` too early (and be left with a dangling pointer) or too late (and leak space).
- A tool such as `valgrind` can detect memory-related faults such as space leaks and out-of-bounds violations. A more specialised tool, *Electric Fence* concentrates on dangling pointers and out-of-bounds violations only.
- Some developers replace the standard implementations of `malloc` and `free` with other versions, for correctness or security reasons.
- Java, Caml, C#, Ruby and Cyclone provide garbage collectors.
- The Boehm collector is a popular and reliable garbage collector.