Annales Mathematicae et Informaticae
48 (2018) pp. 43–50
http://ami.uni-eszterhazy.hu

Bypassing Memory Leak in Modern C++ Realm

Dorottya Papp, Norbert Pataki

Dept. of Programming Languages and Compilers,
Fac. of Informatics, Eötvös Loránd University, Budapest
dorottyapapp@yahoo.com,patakino@elte.hu

Submitted March 5, 2018 — Accepted September 13, 2018

Abstract

Deallocation of dynamically allocated memory belongs to the responsibility of programmers in the C and C++ programming languages. However, compilers do not support the work of the programmers with error or warning diagnostics. Thus the result of this behaviour can be memory leak. Programs' memory consumption may be unreasonably big and even the operating system can be too slow because of the swapping.

We present some different scenarios when memory leak occurs. We show the root cause of the scenarios. This paper presents existing tools for detecting or avoiding memory leak. These tools work in different ways. We analyze the smart pointers of C++11 standard, Valgrind that is a run-time heap profiler, Hans Boehm's garbage collector and the Clang Static Analyzer. We present the pros and cons of the tools. We analyse how difficult it to use these tools, how the efficiency is affected and how these tools can be enhanced for overcome unwanted memory leak. We present our proposals to make the tools more effective.

Keywords: C++, smart pointers, memory leak, garbage collection

MSC: 68N15 Programming languages

1. Introduction

Memory leak can occur in programs written in C/C++ because the deallocation task of dynamically allocated heap memory belongs to the programmers. There is no compiler support for this task and no background job for automatic deallocation. There are pros and cons of this approach. It can be also a problem that memory management of C and C++ is not the same. Programmers use `malloc` and `free` in C programs, but in C++ `new` and `delete` is used typically. However, C's constructs are available in C++, as well. These constructs do not just syntactically differ but have different semantics, so it can be problem if one mixes them up in the source code. However, `new` and `delete` also have versions for one object and for arrays that should not be mixed [2].

However, C++ is an ever evolving language that has been upgraded with new language constructs and new standard libraries in the last years [10]. Furthermore, a bunch of new subtle tools (e.g. static analysers) become available for better development. In this paper we analyse how the modern tools help us bypassing memory leaks. We deal with Valgrind framework that finds memory leak at runtime, and Clang Static Analyzer that is a modern, continuously improving static analyser. We present the C++11's standard smart pointers and the non-standard, but well-known Boehm garbage collector. We analyse what are pros and cons of these constructs.

This paper is organized as follows. We present the tools that are evaluated in this paper in section 2. We present some of our examples that cause memory leak in section 3. We have a large number of test cases to evaluate the tools. We analyse the tools based on the examples and other aspects in section 4. Finally, this paper concludes in section 5.

2. Tools

In this section we present tools that help us overcome memory leaks. The tools work in different ways. We distinguish these tools if they prevent or detect memory leaks.

2.1. Valgrind and Memcheck

Valgrind is a widely-used framework that is able to execute the code with many special validation features and gaining profiling information [7]. This tool executes the code in special "mocked" environment, so the code is untouched, and compiled, linked as usually [8]. However, this safe runtime execution has large overhead, so it cannot be used in production.

Valgrind is a comprehensive tool for detecting problems at runtime, but its primary aim is memory leak detection. It distinguishes different memory leak types:

- Definitely lost: no pointer points to the allocation when the program terminates
- Indirectly lost: no pointer points to that space which were able to access and deallocates the current allocation (e.g. if a root element of a binary tree is

definitely lost than every other nodes in the tree is indirectly lost).

- Still reachable: there is at least one pointer that points to the allocation
- Possibly lost: there is at least one pointer that points to the allocation but it is not exactly the same address that the `new` or `malloc` returns.

2.2. Clang Static Analyzer

Clang is a compiler infrastructure that is based on LLVM [5]. It has many related tools. Clang Static Analyzer uses static analysis and symbolic execution to detect different problems in the code [4]. It can realize divison-by-zero problems, using of uninitialized variables based by examining the source code. As a static analyser it does not execute the code.

The Clang Static Analyzer has three checkers that aim at detecting memory leaks in the source code:

- `unix.MismatchedDeallocator` – searches for incorrect deallocation, when `new`/`delete` and `malloc`/`free` are used together.
- `unix.Malloc` – searches for incorrect `malloc` allocated heap usage (e.g. double `free`, memory leak, etc.)
- `alpha.cplusplus.NewDeleteLeaks` – finds memory leak when the memory is allocated with `new`.

2.3. Boehm Garbage Collector

Many programming languages use garbage collector to ensure the minimalization of memory leak. C/C++ does not offer standard garbage collection (C++11 introduces a minimal ABI [1]). The Boehm garbage collector is able to work in C and C++ programs. It keeps track all variables in the program to check when it can safely execute the deallocation in the background. It uses a modified mark-and-sweep algorithm [3]. It has different interfaces for C and C++-like memory management.

2.4. Smart pointers

The standard smart pointers are able to deallocate memory when the smart pointer objects go out of scope. Smart pointers take advantage of the C++ template construct, so they are independent of the type of the managed memory. C++ template construction is very important feature regarding the performance. Effectiveness of C++ template constructs is still evaluated. The basic operations of smart pointers are those of the raw pointers but smart pointers offer some convenience methods. Different standard smart pointer types are available. However, dealing with memory usage optimization in concurrent execution is still problematic.

The smart pointers are based on the RAII (resource acquisition is initialization) principle: constructors and destructors are automatically executed in a well-defined moment [6]. Invocation of these operations is based on the smart pointer objects lifetime. The major standard smart pointers are `std::unique_ptr<T>`, `std::shared_ptr<T>` and `std::weak_ptr<T>`.

3. Examples

In this section we define examples that are used for evaluation. We have about sixteen use cases. In this section we present some of these. We have analysed how we can modify the examples for garbage collection or smart pointers.

The very first example is a simple one:

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

Valgrind and Clang Static Analyzer detect the memory leak. However, a minimal modification presents the strong limitation of static analysis. If the memory allocation is executed in a function in a different compilation unit then the static analyser does not detect it. This modification does not affect Valgrind because Valgrind works at runtime and does not mind compilation units.

This example can be modified using a smart pointer to avoid memory leak:

```
#include <memory>

int main()
{
  std::unique_ptr<int> p(new int);
}
```

This example can be modified using garbage collector as well. This GC takes advantage of the fact that operator `new` can be overloaded:

```
#include "./gc_cpp.h"

int main()
{
  int * p = new(UseGC) int;
}
```

The following example presents the differences between C and C++ memory routines:

```
#include <cstdlib>

class Vec
{
public:
  Vec() {}
  ~Vec() { delete[] p; }
  void init(int i) { p = new int[i]; }
private:
  int * p;
};

int main()
{
  Vec * vec_pointer_new = new Vec;
  vec_pointer_new->init(5);
  delete vec_pointer_new; // no memory leak

  Vec * vec_pointer_malloc = (Vec*)malloc(sizeof(Vec));
  vec_pointer_malloc->init(5);
  free(vec_pointer_malloc); // memory leak
}
```

The `malloc` and `free` is responsible only for the memory management and cannot deal with C++'s constructs like constructor and destructor. The `new` and `delete` related to objects' lifetime, so these constructs call constructor and destructor, respectively. This can result in memory leak. Valgrind can detect this leak, but Clang Static Analyser does not report it.

One can think that memory leak obviously can be avoided with the help of smart pointers, but it is not true actually:

```
#include <cstdlib>
#include <memory>

int main()
{
  int * p = (int*)malloc(sizeof(int));
  std::unique_ptr<int> u_p(p); // mismatched malloc()/delete
}
```

However, smart pointers offer possibility to pass custom deallocation code snippet but it is not enforced. This problem can be realized at runtime with Valgrind, but cannot be realized with Clang Static Analyser. The major problem is that the C++ standard does not offer functor for `free`. One can develop it, but there is no standard approach for this scenario. On the other hand, functors have other difficulties [9].

Static analysers have an important advantage. These tools do not deal with runtime parameters and are able to check every execution paths. Let us consider the following code snippet:

```
#include <cstdlib>
#include <iostream>

int main()
{
  int * p = new int;

  int input;
  std::cin >> input;

  switch(input)
  {
   case 0: // do something
     break;
   default: // do something else
     delete p; break;
  }
} // memory leak if input==0
```

This potential memory leak is not guaranteed to be checked with Valgrind, but can be detected with Clang Static Analyzer because it does not deal with execution.

4. Evaluation

We have analysed the tools based on the extended set of test cases that we presented in the previous section. The tests revealed the following results:

- Valgrind detects most of the memory leaks in the examples. If the leak is occured on the execution the tool was able to find it. One of the major problems with Valgrind that complex applications are difficult from the viewpoint of execution.

- Clang Static Analyzer finds less memory leak than Valgrind. The major problem is related to cross-translation units and destructor calls. However, this approach does not mind execution paths.

- All memory leaks can be overcome with the garbage collector.

- Smart pointers do a good work in most cases but there are some use cases when smart pointers can also be used erroneously.

After the test cases, we have evaluated the tools based on the following characteristics as well:

- Setup – How difficult is to start the work with this tool
- Documentation – How detailed, well-structured the documentation of the tool is
- Portability – Which platforms can be used
- Further improvements – Is it a mature tool or is it a continuously improving one
- Appreciation – Is it a widespread tool
- Runtime overhead – How the tool affects the runtime
- Memory consumption overhead – How the tool affects the memory consumption
- Compilation time overhead – How the tool affects the compilation time
- False positives – Does the tool report a problem that is not problem actually
- Green field projects or code legacies – Does the tool support big code legacies or is it useful for new projects
- C or C++ support - How the tool is affected by C or C++ code

Our experiences based on the previous characteristics:

- Valgrind is an honored, well-known, widely-used tool. (For instance, during the development of Mozilla Firefox, OpenOffice, MySQL, NASA Mars Exploration Rover, Blender and CMake Valgrind has been used [11].) It is a mature tool, but there are limitations in portability. It does not affect the compilation time, but has overhead, so it cannot be used in production environment. It supports C and C++, as well.
- Clang Static Analyser does not affect the runtime circumstances, but the usage can take long time and has a rather big memory consumption. The documentation is not perfect. It can report false positives. It can be used with C and C++ code, as well. It can be used with code legacies.
- Boehm GC is not a well-documented one, it is hard to set up. It is difficult to use with code legacies. The garbage collector cannot be a standardized one, the community does not support it.
- Smart pointers: well-documented, portable because of the standard. Smart pointers increase the productivity but they cannot work together with pure C.

5. Conclusion

The memory management can be still problematic in C and C++ code. However, there are many tools that can help the programmers to avoid memory leak. We presented some tools for avoiding or detecting memory leaks: static analyser, runtime validation, smart pointers and a garbage collector. We defined a set of test cases to evaluate these tools. After this, we defined other aspects to evaluate and measure the tools' convenience. Based on these we have a comprehensive evaluation of these tools.

References

[1] Boehm, H. J., Spertus, M.: *Garbage collection in the next standard of C++*, in Proc. of the 2009 international symposium on Memory management (2009), pp. 30–38.

[2] Dewhurst, S. C.: "C++ Gotchas Avoiding Common Problems in Coding and Design", Pearson Education (2003).

[3] Edelson, D. R.: *A mark-and-sweep collector C++*, In Proc. of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92) (1992), pp. 51–58.

[4] Horváth, G., Pataki, N.: *Source Language Representation of Function Summaries in Static Analysis*, In Proc. of the 1th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS 2016), Paper Nr. 6.

[5] Lopes, B. C., Auler, R.: "Getting Started with LLVM Core Libraries", Packt Publishing (2014).

[6] Meyers, S.: "Effective Modern C++", O'Reilly (2015).

[7] Nethercote N., Seward J. Valgrind: *A program supervision framework*, Electronic Notes in Theoret, Comput. Sci, **89(2)** (2003), pp. 44–66.

[8] Nethercote N., Seward J. Valgrind: *A framework for heavyweight dynamic binary instrumentation*, in Proc. of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 07), San Diego, (2007), pp. 89–100.

[9] Pataki, N.: *Advanced Functor Framework for C++ Standard Template Library*, Studia Universitatis Babeş-Bolyai, Informatica, **LVI(1)** (2011), pp. 99–113.

[10] Stroustrup, B.: "The C++ Programming Language", Addison-Wesley Publishing Company, Fourth edition (2013).

[11] Valgrind Official Home, `http://valgrind.org/`