**Overview of memory management problems**

One of the primary advantages of C++ is the flexibility and control of resources such as memory it gives to the programmer. This advantage is further amplified by a significant increase in the performance of C++ programs compared to other languages such as Python or Java.

However, these advantages come at a price as they demand a high level of experience from the programer. As Bjarne Stroustrup put it so elegantly:

*"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".*

In this chapter, we will look at a collection of typical errors in memory management that you need to watch out for.

1. **Memory Leaks** Memory leaks occur when data is allocated on the heap at runtime, but not properly deallocated. A program that forgets to clear a memory block is said to have a memory leak - this may be a serious problem or not, depending on the circumstances and on the nature of the program. For a program that runs, computes something, and quits immediately, memory leaks are usually not a big concern. Memory leaks are mostly problematic for programs that run for a long time and/or use large data structures. In such a case, memory leaks can gradually fill the heap until allocation requests can no longer be properly met and the program stops responding or crashes completely. We will look at an example further down in this section.
2. **Buffer Overruns** Buffer overruns occur when memory outside the allocated limits is overwritten and thus corrupted. One of the resulting problems is that this effect may not become immediately visible. When a problem finally does occur, cause and effect are often hard to discern. It is also sometimes possible to inject malicious code into programs in this way, but this shall not be discussed here.

In this example, the allocated stack memory is too small to hold the entire string, which results in a segmentation fault:

**char** str[5];

strcpy(str,"BufferOverrun");

printf("%s",str);

1. **Uninitialized Memory** Depending on the C++ compiler, data structures are sometimes initialized (most often to zero) and sometimes not. So when allocating memory on the heap without proper initialization, it might sometimes contain garbage that can cause problems.

Generally, a variable will be automatically initialized in these cases:

* + it is a class instance where the default constructor initializes all primitive types
  + array initializer syntax is used, such as int a[10] = {}
  + it is a global or extern variable
  + it is defined static

The behavior of the following code is potentially undefined:

**int** a;

**int** b=a\*42;

printf("%d",b);

1. **Incorrect pairing of allocation and deallocation** Freeing a block of memory more than once will cause a program to crash. This can happen when a block of memory is freed that has never been allocated or has been freed before. Such behavior could also occur when improper pairings of allocation and deallocation are used such as using malloc() with delete or new with free().

In this first example, the wrong new and delete are paired

**double** \*pDbl=**new** **double**[5];

**delete** pDbl;

In this second example, the pairing is correct but a double deletion is performed:

**char** \*pChr=**new** **char**[5];

**delete**[] pChr;

**delete**[] pChr;

1. **Invalid memory access** This error occurs then trying to access a block of heap memory that has not yet or has already been deallocated.

In this example, the heap memory has already been deallocated at the time when strcpy() tries to access it:

**char** \*pStr=**new** **char**[25];

**delete**[] pStr;

strcpy(pStr, "Invalid Access");

**Debugging memory leaks with Valgrind**

Even experienced developers sometimes make mistakes that cannot be discovered at first glance. Instead of spending a lot of time searching, it makes sense for C and C++ programmers to use helper tools to perform automatic analyses of their code.

In this section, we will look at *Valgrind*, a free software for Linux and Mac that is able to automatically detect memory. Windows programers can for example use the Visual Studio debugger and C Run-time Library (CRT) to detect and identify memory leaks. More information on how to do this can be found here: [**Find memory leaks with the CRT Library - Visual Studio | Microsoft Docs**](https://docs.microsoft.com/en-us/visualstudio/debugger/finding-memory-leaks-using-the-crt-library?view=vs-2019)

With recent versions of MacOS, occasional difficulties have been reported with installing Valgrind. A working version for MacOS Mojave can be downloaded from GitHub via Homebrew: [**GitHub - sowson/valgrind: Experimental Version of Valgrind for macOS 10.14.6 Mojave**](https://github.com/sowson/valgrind)

Valgrind is a framework that facilitates the development of tools for the dynamic analysis of programs. Dynamic analysis examines the behavior of a program at runtime, in contrast to static analysis, which often checks programs for various criteria or potential errors at the source code level before, during, or after translation. More information on Valgrind can be found here: **[Valgrind: About](http://valgrind.org/info/" \t "_blank)**

The Memcheck tool within Valgrind can be used to detect typical errors in programs written in C or C++ that occur in connection with memory management. It is probably the best-known tool in the Valgrind suite, and the name Valgrind is often used as a synonym for Memcheck.

The following code generates a memory leak as the integer array has been allocated on the heap but the deallocation has been forgotten by the programmer:

**int main()**

**{**

**int \*pInt = new int[10];**

**return 0;**

**}**

The array of integers on the heap to which pInt is pointing has a size of 10 \* sizeof(int), which is 40 bytes. Let us now use Valgrind to search for this leak.

After compiling the memory\_leaks\_debugging.cpp code file on the right to a.out, the terminal can be used to start Valgrind with the following command:

valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --log-file=/home/workspace/valgrind-out.txt /home/workspace/a.out

Let us look at the call parameters one by one:

* **--leak-check** : Controls the search for memory leaks when the client program finishes. If set to summary, it says how many leaks occurred. If set to full, each individual leak will be shown in detail.
* **--show-leak-kinds** : controls the set of leak kinds to show when —leak-check=full is specified. Options are definite, indirect, possible reachable, all and none
* **--track-origins** : can be used to see where uninitialised values come from.

You can read the file into the terminal with:

cat valgrind-out.txt

In the following, a (small) excerpt of the valgrind-out.txt log file is given:

==952== 40 bytes **in** 1 blocks are definitely lost **in** loss **record** 18 **of** 45

...

==952== **by** 0x10019A377: operator new(unsigned long) (**in** /usr/lib/libc++abi.dylib)

...

==952== **by** 0x100000F8A: main (memory\_leaks\_debugging.cpp:12)

...

==952== LEAK SUMMARY:

==952== definitely lost: 40 bytes **in** 1 blocks

==952== indirectly lost: 0 bytes **in** 0 blocks

==952== possibly lost: 72 bytes **in** 3 blocks

==952== still reachable: 200 bytes **in** 6 blocks

==952== suppressed: 18,876 bytes **in** 160 blocks

As expected, the memory leak caused by the omitted deletion of the array of 10 integers in the code sample above shows up in the leak summary. Additionally, the exact position where the leak occurs in the code (line 12) can also be seen together with the responsible call with caused the leak.

This short introduction into memory leak search is only an example of how powerful analysis tools such as Valgrind can be used to detect memory-related problems in your code.